# CS170 Review

## Kernels and processes

### Four fundamental OS concepts

- **Thread**: single unique execution context; program counter, registers, execution flags, stack
- **Address space with translation**: programs execute in an address space that is distinct from the memory space of the physical machine
- **Process**: an instance of an executing program is a process consisting of an address space and one or more threads of control; a process in memory includes a program counter, stack and heap, and data/instruction section (PCB)
- **Dual mode operation**: onely the system has the ability to access certain resouces; the OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from the program virtual addresses to machine physical addresses
  - User to kernel: sets system mode and saves the user PC; OS code carefully puts aside user state then performs the necessary operations
  - Kernel to user: user transition clears system mode and restores appropriate user PC; user program returns from interrupt

To run a program, the OS loads code and data segments of executable file into memory, creates stack and heap, trnasfers control to it and provides ervices to it. As a process executes, it changes **state**:

- **New**: the process is being created
- **Running**: instructions are being executed
- **Waiting**: the process is waiting for some event to occur
- **Ready**: the process is waiting to be assigned to a processor
- **Terminated**: the process has finished execution

**Process control block** (PCB) stores information associated with each process: process state, program counter, copy of CPU registers, memory-management information (page table), accounting information, IO status information, etc.

When CPU switches to another process with a new address space, the system must save the state of the old process and load the saved state for the new process via a **context switch**. The context of a process represented in the PCB.

### Process creation

- Parent process create child processes, identified via a process id (pid)
- Options in resource sharing: parent and children share all resources; children share subset of parent's resources; parent and children share no resources
- Parent and children execute concurrently; parent watis until children terminate (by join)
- Options in address space: child duplicate of parent, or has another program being loaded
- In UNIX: `fork` creates new process with duplicated address space with a copy of some code and data; `exec` used after a `fork` to replace the process' memory space with a new program
- Summary of `fork`
  - creates a child PCB with new pid
  - allocates memory for the child and duplicate everything from parent, including text/heap/stack, but notice child space is a copy of parent space; all fds that are open in hte parent are duplicated in the child (file `read()/write()` set offsets are shared); many environmental setings and ahred segments are inherited by child (parent and child can communicate through shared segments)
  - add the child process to ready queue

```c
int main() {
    int pid = fork();
    if (pid < 0) { exit(-1); } // error occured
    else if (pid == 0) { execlp("/bin/ls", "ls", NULL); } // child proces
    else { wait(NULL); exit(0); } // parent process
}
```

### Process termination

- Process executes last statement and asks the OS to delete it (exit)
- Output data from child to parent (via wait), process resources are deallocated
- Parent may terminate children processes: task assigned to child is no longer required of if parent exits

### UNIX signals

- A event similar to hardware interrupt without priorities
- Used to informa a user process of an event, e.g., user pressed delete key
- Each signal is represented by a numeric value: e.g., SIGINT(2), SIGKILL(9), etc.
- A UNIX signal is raised by a process using a system call `kill(signal, pid)` or a shell command `kill -s signal pid`
- Each signal is maintained as a single bit in the process table entry of the receving process: the bit is set when the corresponding signal arrives (no waiting queues)
- A signal is processed as soon as the process enters in user mode
- 3 ways to handle a signal
  - ignore it: `signal(signum, SIG_IGN)`
  - run the default handler: `signal(signum, SIG_DFL)`
  - run a user-defined handler: `signal(signum, handler)`

```c
#include <signal.h>
void cnt(int sig) {
    static int count = 0;
    if (count == 1) signal(SIGINT, SIG_DFL);
}
int main() {
    signal(SIGINT, cnt);
    while(1);
}
```

### UNIX pipes for IPC

- The pipe interface is intended to look a file interface
- `pipe(fd)` creates the pipe and kernel allocates a buffer with two pointers `fd[0]` to read and `fd[1]` to write
- pipe handles are copied on `fork` (just like a usual fd)
- `ls|wc`
  - process1 `dup2(fd[1], 1); close(fd[0]); execvp("ls, ...")`
  - process2 `dup2(fd[0], 0); close(fd[1]); execvp("wc", ...)`

```c
int fd[2]; pipe(fd); // fd[0] is read end, fd[1] is write end
if (fork() == 0) { read(fd[0], buffer, size); } // child process
else { write(fd[1], "hello", size); } // parent process
```

## Concurrency and threads

A thread is a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

- Shared state: heap, global variables, code
- Per-thread state: thread control block (TCB), stack information, saved registers, thread metadata
- Benefits: responsiveness, resource sharing, economy, scalability
- Thread abstraction: infinite number of processors, threads execute with variable speed
- Programs must be designed to work with any schedule
- Two threads run concurrently if their logical flows overlap in time; otherwise, they are sequential
- Same as process: each has its own logical control flow, each can run concurrently, each is context switched
- Different from process: threads share code and data while proceses (typicall) do not, threads are somewhat cheaper than processes with less overhead

### POSIX threads (pthreads) interface

- Creating and join threads: `pthread_create`, `pthread_join`
- Determining your tid: `pthread_self`
- Terminating threaads: `pthread_cancel`, `pthread_exit`
  - `exit` terminates all threads; `return` terminates current thread

```c
#include <pthread>
void *PrintHello(void *id) { printf("..."); }
int main() {
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, PrintHell, (void*) 0);
    pthread_create(&thread1, NULL, PrintHell, (void*) 1);
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
}
```

### Nachos threads

- `Thread(char *name)`: create a thread
- `Fork(VoidFunctionPtr func, int arg)`: place this thread in a ready queue for executing a function
  - Allocate thread control block (TCB)
  - Allocate and setup stack: build stack frame for base of stack; put func and args on stack
  - Put thread on ready list, and it will run sometime later
  - `stub(func, args) { (*func)(args); thread_exit(); }`
- `Yield()`: suspend the calling thread and the system selects a new ready one for execution
- `Sleep()`: suspend the current thread, change its sate to BLOCKED, and remove it from the ready list
- `Finish()`, `~Thread()`

### Kernel threads vs user-level threads

- Kernel threads are recognized and supported by the OS kernel; OS explicitly performs scheduling and context switching of kernel threads
- User-level thread management done by user-level threads library
  - OS kernel does not know/recognize there are multiple threads running in a user program
  - The user program (library) is responsible for scheduling and context switching of its threads

## Process/thread synchronization

### Synchronization problem

- Race condition: two or more processes (threads) are reading and writing on shared data and the final result depends on who runs precisely and when
- Critical section: the part of the program where shared variables are accessed
- Deadlock: two or more threads (or processes) are waiting indefinitely for an event that can be only caused by one of these waiting processes
- Starvation: indefinite blocking. Deadlock is starvation but not vice versa

### Property of cirtical-section solution

- Mutual exclusion: only one can enter the critical section, providing safety
- Progress: if some processes wish to enter their critical section and nobody is in the critical section, then one of them will enter in a limited time, providing liveness
- Bounded waiting: if one process starts to wait for entering an critical section, there is a limit on the number of times other processes entering the section before this process enters, providing liveness
- Locks, semaphore, condition variables

### Lock prevents someone from doing something

- `void Acquire()` atomically waits until the lock is unlocked, and then sets the lock to be locked.
- `void Release()` atomically changes the state to be unlocked. Only the thread who owns the lock can release.
- Thread waits if there is a lock.
- It enters the critical after acquiring a lock.
- Only the thread who locks can unlock.
- Lock can be in one of two states: locked or unlocked
- Typically associate a lock with a piece of shared data for mutual exclusion. When a thread needs to access, it first acquires the lock, and then accesses data. Once access completes, it releases the lock

### Lock implementation

- Atomic load/store are complex and error prone
- Alternatively, interrupt disabling/enabling and hardware primitives; avoid context-switching by
  - preventing external events by disabling interrupts
  - Avoid internal events
- But interrupt disabling/enabling doesn't work/scale well on multiprocessor
- ALternatively, atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
- Examples of atomic instructions
  - `TestAndSet(&address)` fetch old value, set it to 1
  - `Swap(&address, register)` swap value in the address and register
  - `CompareAndSwap(&address, reg1, reg2)` sets address to reg2 if it equals reg1
- Spin lock using `TestAndSet`: mutual exclusion but not bounded waiting, busy waiting
  - shared boolean variable as lock: true means it is acquired by others; initialized to false
  - acquire: `while(TestAndSet(&lock))`
  - release: `lock = false`
  - Bette with sleep: minimizes busy waiting, only busy-waiting to atomically check lock value
  - pro: machine can receive interrupts; work on a multiprocessor
  - con: inefficient spin lock; no guarantee on bounded waiting

```
int guard = false; int value = free;
Acquire() {
    while(test&set(guard));
    if (value == busy) {
        put thread on waiting queue
        sleep and set guard to false
    } else {
        value = busy
        guard = false
    }
}
```

```
Release() {
    while(test&set(guard));
    if anyone on waiting queue {
        take thread off waiting queue
        place it on ready queue
    } else {
        value = free
    }
    guard = false
}
```

### Semaphore

- `void P()` atomically waits until the counter is greater than 0 and then decreases the counter
- `void V()` atomically decreases the counter
- It uses a nonnegative integer variable, can only be accessed/modified via two individual (atomic) operations
  - `wait(S) { while S <= 0: wiating in a queue; S--; }`
  - `signal(S) { wakeup some waiting thread; S++; }`
- Counting semaphore: initial value representing how many threads can be in the critical section
- Binary semaphore (aka mutex lock): integer value ranged between 0 and 1
- Producer thread: `space->P(), data->V()`
- Consumer thread: `data->P(), space->V()`

### Condition variable

- `void Wait(Lock *myLock)` atomically releases the lock and waits. When it is returned, the lock is reacquired again.
- `void Signal(Lock *myLock)` wake up one waiting thread to run. The lock is not released.
- `void Broadcast(Lock *myLock)` wake up all threads waiting on the condition. The lock is not released.
- Hoare-style scheduling
  - Signaler gives lock, CPU to waiter; waiter runs immediately
  - Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
- Mesa-style shceduling
  - Signaler keeps lock and continues to process
  - Waiter placed on ready queue with no special priority
  - Practically, need to check condition again after wait

### Design advices for synchronization

- Allocate one lock for each shared variable or a group of shared variables
- Add condition variables (waking up a thread is triggered by some condition)
- Fine-grain: More performance flexibility and scalability but possibly more design complexity and synchronization overhead
- Coarse-grain: Easy to ensure correctness but possibly difficult to scale

### Read-writer lock

- `pthread_rwlock_rdlock()`: multiple readers can acquire the lock if no writer holds the lock.
- `pthread_rwlock_wrlock()`: the calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock rwlock. Otherwise, the thread blocks until it can acquire the lock. Writers are favored over readers of the same priority to avoid writer starvation.
- Reader: wait until no writers; access database; checkout (wake up a waiting writer)
- Writer: wait until no active readers or writers; access database; checkout (wakeup waiting readers or writer)

## Main memory and address translation

### How to translate address

- Contiguous memory allocation
- Segmentation
- Address tarnslation with paging
- TLB support and performance impact

### Objective of memory management: run programs

- Get CPU cycle and allocate memory
- Load instruction and data segments of executable file into memory
- Create stack and heap
- Set the starting address and execute
- Provide services to it

### Role of memory management

- Manage data/instructions in memory in order to execute
- Keep track of which parts of memory are currently being used by whom
- Decide which processes (or parts thereof) and data to move into and out of memory
- Allocate and deallocate memory space as needed

### Logical vs. physical address space

- Logical (virtual) address: generated by the CPU
- Physical address: address seen by the memory unit
- Same in compile-time and load-time address-binding schemes
- Differ in execution-time address-binding scheme

### Binding of instructions and data to memory

- Compile time (gcc): If memory location known a priori, absolute code can be generated
- Load time (ld): Must generate relocatable code if memory location is not known at compile time
- Execution time (dll): Binding delayed until run time if the process can be moved during its execution from one memory segment to another.

### Address space translation

- Address space
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Program operates in an address space that is distinct from the physical memory space of the machine
- CPU issues virtual address to MMU which translates to physical memory to memory
- Translation provides protection and process address space isolation. With translation, every program can be linked/loaded into same region of user address space

### Allocation method: Contiguous allocation

- Main memory has two partitions: Resident OS usually held in low memory; user processes then held in high momery
- Hardware support and protection
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses (each logical address must be less than the limit register)
  - MMU maps logical address dynamically
- During switch, kernel loads new base/limit from PCB
- Issues with simple R&B method
  - **External fragment** problem (free gaps between used slots): Not every process is the same size; over time, memory space becomes fragmented
  - Missing support for sparse address space: Would like to have multiple chunks/program, e.g., code, data, stack
  - Hard to do inter-process share

### Allocation method: Segmentation

- Memory-management scheme that supports user semantic view of memory
- A program is a collection of segments
- A segment is a logical unit such as code, data, stack (can be shared)
- Each segment is given region of contiguous memory: base and limit, can reside anywhere in physical memory
- CPU issues a virtual address `(seg, offset)`, if `offset < ST[seg].limit`, then the physical address is `ST[seg].base + offset`; otherwise raises an exception
- Issues with segmentation
  - High overhead of managing a large number of variable size and dynamically growing memory: Memory is divided into many used/unused regions over time; not easy to find space to fit for a new segment
  - External fragmentation: Free gaps between allocated chunks
  - Internal fragmentation: Don't need all memory within allocated chunks

### Allocation method: Paging

- Divide physical memory into fixed-sized blocks called frames or pages
- Divide logical memory into blocks of same size called pages or logical pages
- Translation conducted by page table which is kept in main memory
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PRLR) indicates size of the page table
- Useful constants: 2^10 = 1K, 2^20 = 1M, 2^30 = 1G, 2^40 = 1T
- A page table per process is needed to translate logical to physical addresses
- We can use a vector of bits to represent availability of each page
- Address generated by CPU is divided into a page number (p) and a page offset

d.
  - Offset from virtual address copied to physical address
  - Virtual page number is all remaining bit, the translated physical page number is copied from page table into physical address
  - Physical address = `PageTable[LogicalPageNumber] + PageOffset`

- Each page is associated with permission bits (e.g., valid, read, write, etc.)
- Page table size is determined by the number of pages in virtual space

### TLB for faster address translation

- Every data/instruction access requires two memory access. Speed can be improved by using a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
- The TLB stores the recent translations (i.e., entries) of virtual memory to physical memory and can be called an address-translation cache.
- Address translation for logical page p
  - If p is in associative register, get physical page number out
  - Otherwise get frame number from page table in memory
- Typical TLB: 8~4096 entries; 0.5~1 clock cycle to access; 10~100 cycles miss penalty; 0.01~10% miss rate
- Effective (average) address translation cost: cost(TLB lookup) + cost(full translation) * TLB miss rate
- Total memory access cost: cost(avg address translation) + cost(memory access)
- Memory control infomation are stored infomation in
  - Thread control block (TCB): stack info, registers and machine state
  - Process control block (PCB): process id data, state data, control data (memory space management, pointer to a page table)

### Shared pages through paging

- Shared code
  - One copy of read-only code shared among processes
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
- Copy-on-write (COW): Lazy copy during process creation
  - Optimization of Unix system call `fork`: A child process copies address space of parent. Most of time it is wasted as the child performs `exec`.
  - COW allows both parent and child processes to initially share the same pages in memory. A shared page is duplicated only when modified. It allows more efficient process creation as only modified pages are copied.

### Page table entry (PTE)

- Used to memorize a page is shared, detect need for duplication
- Invalid PTE can imply different things: region of address space is actually invalid or the page/directory is just somewhere else than memory
- Validaty checked first: OS can use other bits for location info
- COW uses PTE which contains bit indicating a page is shared with a parent
- Demand paging keeps only active pages in memory, place others on disk and mark their PTEs invalid
- x86 PTE
  - Address format: (10, 10, 12-bit offset)
  - Intermediate page tables called directories: (20-bit page frame number, 12-bit flags)
- Zero fill on demand
  - Security and performance advantages: New pages carry no information
  - Give new pages to a process initially with PTEs marked as invalid
  - During access time, page fault causes physical frames to be allocated and filled with zeros
  - Often, OS creates zeroed pages in background

### One-level page table

- Maximum size of logical space = # entry * page size
- Each page table needs to fit into a physical memory page because a page table needs consecutive space. Memory allocated to a process is a sparse set of nonconsecutive pages.
- One-level page table cannot handle large space, for example
  - 32-bit address space with 4KB per page
  - Page table would contain $2^{32/2}12$ = 1 million entries
  - 4B per entry: Need a 4MB page table with contiguous space (impossible)
  - Only 4KB / 4B = 1K entries, hence 1K * 4KB = 4MB logical space
- Pro: Simple memory allocation, easy to share
- Con: Cannot handle a large (sparse) virtual address space, e.g., on Unix, code starts at 0, stack starts at 2^32-1
- Con: Not all pages are used all the time. It would be nice to have working set of page table in memory

### Two-level page table

- Address translation scheme: Logical address is of the form (p1, p2, d), where p1 is an index into the outer level-1 page table, p2 is the displacement within the page of the level-2 inner page table
- Tables are of fixed size (1K entries, 4K per entry, 4KB in total)
- Valid bits on PTE: Not every level-2 table is needed. Even when exist, level-2 tables can reside on disk if not in use
- Design consideration on paging
  - Bits of d = log (page size)
  - Bits of p1 >= log (# entries in level-1 table)
  - Bits of p2 = log (# entries in level-2 table)
  - # physical pages <= 2^(PTE size of level-2 table)
  - Physical space size = # physical pages * page size
  - Logical space size = # entry in level-1 table * # entry in level-2 table * page size
- Example: A logical address (32-bit machine with 4KB page size) is divded into
  - A page number area with 20 bits, a page offset consisting of 12 bits
  - Each PTE uses 4B: Hence it is divided into 10-10-12-bit
  - Maximum logical space size: 1K * 1K * 4KB = 4GB
  - What if we use 2B for each PTE?
    - Logical space size remains the same (0.5K * 2K * 4KB)
    - Physical space decreases (2^16 * 4K)

### Three-level page table

- Design consideration on paging
  - Bits of d = log (page size)
  - Bits of p1 >= log (# entries in level-1 table)
  - Bits of p2 = log (# entries in level-2 table)
  - Bits of p3 = log (# entries in level-3 table)
  - # physical pages <= 2^(entry size of level-3 table)
  - Physical space size = # physical pages * page size
  - Logical space size = # entry in level-1 table * # entry in level-2 table # *entry in level-3 table* page size

### Hashed page table

- Common in address spaces > 32bits
- Size of page table grows proportionally as large as amount of virtual memory allocated to processes
- Use hash table to limit the cost of search
  - To one, or at most a few, page-table entries
  - One hash table per process
  - This page table contains a chain of elements hasing to the same location
- Use this hash table to find the physical page of each logical page: If a match is found, the corresponding physical frame is extracted

### Inverted page table

- One hash table for all processes
- One entry for each real page of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

### Summary

| | Advantages | Disadvantages |
|---|---|---|
| Segmentation | Fast context switching: Segment mapping maintained by CPU | External fragmentation |
| Paging (single level) | No external fragmentation, fast easy allocation | Large table size ~ virtual memory, internal fragmentation |
| Paged segmentation, two-level paging | Table size ~ # of pages in virtual memory, fast easy allocation | Multiple memory references per page access |
| Inverted table | Table ~ # of pages in physical memory | Hash function more complex |

- Page tables
  - Memory divided into fixed-size chunks of memory
  - Virtual page number from virutal address mapped through page table to physical page number
  - Offset of virtual address is same as physical address
  - Large page tables can be placed into virtual memory
- Useage of PTE: page sharing, copy on write, page on demand, zero fill on demand
- Multiple page tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table: Size of page table related to physical memory size

## Nachos

### Nachos thread operations

- `Thread()` creates a thread
- `Fork(VFP func, int arg)` lets a thread execute a function
- `Yield()` suspends the calling thread and select a new one for execution
- `Sleep()` suspends the current thread, changes its state to BLOCKED, and removes it from the ready list
- `Finish()`

### Code execution in Nachos

- User mode executes instructions which only access the user space.
- Kernel mode executes when Nachos first starts up or when an instruction causes a trap, e.g., illegal instruction, page fault, or system call, etc.
- Load instructions into the machine's memory
- Initializes registers (PC, etc.)
- Tell the machine to start executing instructions.
- The machine fetches the instruction, decodes it, and executes it.
- Repeats until all instructions are executed.
- Handle interrupt/page fault if necessary.

### Scheduler object

- Decide which thread to run next.
- Invoked when the current thread gives up CPU.
- The current Nachos shceduling polycy is round-robin: slecting the front of ready queue list, appending new threads to the end

### Machine object: implements a MIPS machine

- An instance created when Nachos starts up.
- Supported public variables: 40 registers, 4KB memory (32 pages), single linear page table virtual memory.

### Interrupt object: maintains an event queue with simulated clock

- `SetLevel(IntStatus level)` is used to temporarily disable and re-enable interrupts.
- `OneTick()` advances 1 clock tick.
- `CheckIfDue(bool advanceClock)` examines if some event should be serviced.
- `Idle()` advances the clock to the time of the next scheduled event.

### Timer object

- Generates interrupts at regular or random intervals, then Nachos invokes the predefined clock event handling procedure.

### Noff: binary code format

- A Noff-format file contains
  - The Noff header, describing the contents of the rest of the file
  - TEXT: Executable code segment.
  - DATA: Initialized data segment.
  - BSS: Uninitialized data segment: stagically-allocated variables
- Each segment has the following information
  - virtualAddr: virtual address that segment begins at
  - inFileAddr: pointer within the Noff file where that scetion actually begins
  - size (in bytes) of that segment

### Nachos thread: user process for executing a program

- A Nachos thread is extended as a process
- Each process has its own address space containing executable code (TEXT), initialized data (DATA), uninitialized data (BSS), and stack space for fucntion call/local variables
- Each addrspace is as big as `code.size + data.size + bss.size + userStackSize`
- A process owns some other objects, such as open file descriptors
- Steps in user process creation
  - Create an address space
  - Zero out all of physical memory
  - Read the binary into physical memory and initialize data segment
  - Initialize the translation tables to do a one-to-one mapping between virtual and physical address
  - Zero all registers, setting PCReg, NextPCReg to 0 and 4 respectively
  - Set the stackpointer to the largest virtual address of the process
- Actions of `nachos -x halt`
  - The main thread starts by running function `StartProcess()` in `progtest.cc`. This thread is used to run halt binary.
  - `StartProcess()` allocates a new address space and loads the halt binary. It also initializes registers and sets up the page table.
  - `Machine::Run()` executes the halt binary using the MIPS emulator: The halt binary invokes the system call `Halt()`, which causes a trap back to the Nachos kernel via functions `RaiseException()` and `ExceptionHandler()`.
  - The execution handler determines that a `Halt()` system call was requested from user mode, and it halts Nachos.

### Thread execution flow when running Nachos under thread directory

When the *threads* version of NACHOS is started, `main()` in file `main.cc` under threads directory performs the following key operations.

1. Call `Initialize()` in file `system.cc` under threads directory to initialize Nachos global data structure. For example, set random yield or not based on parameters, and create an interrupt object and a scheduler object etc.
2. Call `ThreadTest()` in the file `threadtest.cc` under threads directory. This creates two Nachos threads to execute. This calls two function methods `Thread::Fork()` and `Thread::Yield()`.
3. Call `currentThread->Finish()` in file `thread.cc` which calls `Thread:Sleep()`, similar to `Thread:Yield()`. If the procedure "main" returns, then the program "nachos" will exit (as any other normal program would). But there may be other threads on the ready list. We switch to those threads by saying that the "main" thread is finished, preventing it from returning.

```
DebugInit(debugArgs);                           // initialize DEBUG messages
stats = new Statistics();                       // collect statistics
interrupt = new Interrupt;                      // start up interrupt handlin
scheduler = new Scheduler();                    // initialize the ready queue

currentThread = new Thread("main");

currentThread->setStatus(RUNNING);

interrupt->Enable();
CallOnUserAbort(Cleanup);                       // if user hits ctl-C
```

`Thread::Fork()` is defined in `thread.cc` under threads directory. It calls `StackAllocate(func, arg)` defined in `thread.cc` under threads directory. This function allocates stack space and initialize the thread control block and the register state information. Notice that "func" address is assigned to `InitialPCState` slot of the thread control block. This address will be used as the entrance point when this thread starts to execute, which will be discussed in two assembly functions `SWITCH()` and `ThreadRoot()` below.

```
//StackSize is 4K defined in thread.h
stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
machineState[PCState] = (int) ThreadRoot;
machineState[StartupPCState] = (int) InterruptEnable;
machineState[InitialPCState] = (int) func;
machineState[InitialArgState] = arg;
machineState[WhenDonePCState] = (int) ThreadFinish;
```

`ThreadTest()` in the file `threadtest.cc` calls `currentThread->Yield()` to yield CPU to another ready thread and Nachos schedules another thread for execution. `Thread::Yield()` is defined in `thread.cc` under threads directory.

```
nextThread = scheduler->FindNextToRun();
if (nextThread) { scheduler->ReadyToRun(this); scheduler->Run(nextThread)
```

1. `FindNextToRun()` method defined in `scheduler.cc` under threads directory finds next thread in the ready queue to run. That essentially removes the first one in the ready queue.
2. `ReadyToRun()` in `scheduler.cc` under threads directory marks the selected thread to be ready and append it to the end of ready list.
3. `Run()` in `scheduler.cc` under threads directory executes the selected thread. It does context switch from old thread to new thread by calling assembly code `SWITCH(oldThread, nextThread)` in `switch.s` under threads directory. `SWITCH()` shifts execution from the old thread to new thread. It saves register state for the old thread and loads the new data from the new thread's control block. For example, if the host hardware is MIPS (actually not as we run on the Intel chip. I use MIPS because its instructions are easy to read.)

```
    # a0 -- pointer to old Thread
    # a1 -- pointer to new Thread

SWITCH:
    sw  sp, SP(a0)      # save new stack pointer
    sw  s0, S0(a0)      # save all the callee-save registers
    sw  s1, S1(a0)
    sw  s2, S2(a0)
    sw  s3, S3(a0)
    sw  s4, S4(a0)
    sw  s5, S5(a0)
    sw  s6, S6(a0)
    sw  s7, S7(a0)
    sw  fp, FP(a0)      # save frame pointer
    sw  ra, PC(a0)      # save return address

    lw  sp, SP(a1)      # load the new stack pointer
    lw  s0, S0(a1)      # load the callee-save registers
    lw  s1, S1(a1)
    lw  s2, S2(a1)
    lw  s3, S3(a1)
    lw  s4, S4(a1)
    lw  s5, S5(a1)
    lw  s6, S6(a1)
    lw  s7, S7(a1)
    lw  fp, FP(a1)
    lw  ra, PC(a1)      # load the return address

    j   ra
    .end SWITCH

ThreadRoot:
    jal StartupPC       # call startup procedure
    move a0, InitialArg
    jal InitialPC       # call main procedure
    jal WhenDonePC      # when were done, call clean up procedure
```

Constants `SP`, `S0`, ..., `PC` etc are defined in `switch.h` under threads directory. For example, `SP=0` for MIPS host. Notice that `0` means the first position `stackTop` in the thread control block which is defined in `thread.h` What does `SWITCH()` finally call after saving and loading context? Namely what does the return address "ra" point to before executing "j ra"? Notice that `StackAllocate(func, arg)` has assigned

```
machineState[PCState] = (int) ThreadRoot;   # in MIPS, PCState = PC/4-1 =
machineState[InitialPCState] = (int) func;  # in MIPS, InitialPCState = S
```

Thus `SWITCH()` calls `j ra` which is `ThreadRoot`. Then `ThreadRoot()` will call "func" saved and loaded from `InitialPCState`.

### Scheduler

- `ReadyToRun`. Mark a thread as ready, but not running. Put it on the ready list, for later scheduling onto the CPU.
- `FindNextToRun`. Return the next thread to be scheduled onto the CPU. If there are no ready threads, return NULL. Thread is removed from the ready list.
- `Run`. Dispatch the CPU to nextThread. Save the state of the old thread, and load the state of the new thread, by calling the machine dependent context switch routine, `SWITCH`. Note: we assume the state of the previously running thread has already been changed from running to blocked or ready (depending). The global variable currentThread becomes nextThread.

### Thread

Data structures for managing threads. A thread represents sequential execution of code within a program. So the state of a thread includes the program counter, the processor registers, and the execution stack. Stack size is fixed.

- `Thread`. Initialize a thread control block, so that we can then call `Fork`.
- `~Thread`. De-allocate a thread. NOTE: the current thread *cannot* delete itself directly, since it is still running on the stack that we need to delete. NOTE: if this is the main thread, we can't delete the stack because we didn't allocate it -- we got it automatically as part of starting up Nachos.
- `Fork`. Invoke `(*func)(arg)`, allowing caller and callee to execute concurrently. NOTE: although our definition allows only a single integer argument to be passed to the procedure, it is possible to pass multiple arguments by making them fields of a structure, and passing a pointer to the structure as "arg". Implemented as the following steps: (1) Allocate a stack,

2. Initialize the stack so that a call to `SWITCH` will cause it to run the procedure, (3) Put the thread on the ready queue.

- `Finish`. Called by `ThreadRoot` when a thread is done executing the forked procedure. NOTE: we don't immediately de-allocate the thread data structure or the execution stack, because we're still running in the thread and we're still on the stack! Instead, we set `threadToBeDestroyed`, so that `Scheduler::Run()` will call the destructor, once we're running in the context of a different thread. NOTE: we disable interrupts, so that we don't get a time slice between setting threadToBeDestroyed, and going to sleep.
- `Yield`. Relinquish the CPU if any other thread is ready to run. If so, put the thread on the end of the ready list, so that it will eventually be re-scheduled. NOTE: returns immediately if no other thread on the ready queue. Otherwise returns when the thread eventually works its way to the front of the ready list and gets re-scheduled. NOTE: we disable interrupts, so that looking at the thread on the front of the ready list, and switching to it, can be done atomically. On return, we re-set the interrupt level to its original state, in case we are called with interrupts disabled. Similar to `Thread::Sleep()`, but a little different.
- `Sleep`. Relinquish the CPU, because the current thread is blocked waiting on a synchronization variable (Semaphore, Lock, or Condition). Eventually, some thread will wake this thread up, and put it back on the ready queue, so that it can be re-scheduled. NOTE: if there are no threads on the ready queue, that means we have no thread to run. `Interrupt::Idle` is called to signify that we should idle the CPU until the next I/O interrupt occurs (the only thing that could cause a thread to become ready to run). NOTE: we assume interrupts are already disabled, because it is called from the synchronization routines which must disable interrupts for atomicity. We need interrupts off so that there can't be a time slice between pulling the first thread off the ready list, and switching to it.
- `StackAllocate`. Allocate and initialize an execution stack. The stack is initialized with an initial stack frame for `ThreadRoot`, which: (1) enables interrupts, (2) calls `(*func)(arg)` and (3) calls `Thread::Finish`.
- `SaveUserState`. Save the CPU state of a user program on a context switch.
- `RestoreUserState`. Restore the CPU state of a user program on a context switch.

## MIPS

### Calling convention

- `$0`. Zero register.
- `$1` - `$at`. The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
- `$2–$3` - `$v0–$v1`. Used for return values of a subroutine.
- `$4–$7` - `$a0–$a3`. Used for arguments of a subroutine.
- `$8–$15, $24, $25` - `$t0–t9`. Temporary registers.
- `$16–$23` - `$s0–$s7`. Saved registers.
- `$26–27` - `$k0–$k1`. Kernel reserved registers. Do not use.
- `$28` - `$gp`. Globals pointer used for addressing static global variables.
- `$29` - `$sp`. Stack pointer.
- `$30` - `$fp ($s8)`. Frame pointer.
- `$31` - `$ra`. Return address in a subroutine call.

Caller and callee save

- Callee saves
  - a procedure clears out some registers for its own use
  - register values are preserved across procedure calls
  - MIPS calls these saved registers, and designates `$s0–$s8` for this useage
  - the called procedure saves register values in its AR, uses the registers for local variables, restores register values before it returns.
- Caller saves
  - the calling program saves the registers that it does not want a called procedure to overwrite
  - register values are NOT preserved across procedure calls
  - MIPS calls these temporary registers, and designates `$t0–$t9` for this useage
  - procedures use these registers for local variables, because the values do not need to be preserved outside the scope of the procedure.

## Virtual memory

### Demand paging

- Virtual memory can be much larger than physical memory.
  - Combined memory of running processes much larger than physical memory.
  - More programs fit into memory, allowing more concurrency.
- Supports flexible placement of physical data. Data could be on disk or somewhere across network.
- Variable location of data transparent to user program. Performance issue, not correctness issue.
- Bring a page into memory only when it is needed.
  - Less I/O. Less memory. Faster response. More users supported.
- Valid bits in a PTE.
  - Valid means in-memory. Invalid means not-in-memory.
  - Initially valid bit is set to invalid to on all PTEs.
  - Not in memory causes page fault.
- Dirty bit means this page has been modified. It needs to be written back to disk.
- Steps in handling a page fault.
  - Reference a page.
  - Page fault exception.
  - Page is on backing store.
  - Bring in missing page.
  - Reset page table.
  - Restart instruction.
- What does the OS do on a page fault?
  - Choose an old page to replace. If old page is modified (dirty is set), write contents back to disk. Chagne its PTE and any cached TLB to be invalid.
  - Get an empty physical page. Load new page into memory from disk. Update PTE, invalidate TLB for new entry.
  - Continue thread from original faulting location. Restart the instruction that caused the page fault.
- Performance of demand paging.
  - Page fault rate = $p$. If $p = 0$, no page fault. If $p = 1$, every reference is a fault.
  - Effective access time (EAT) = (1-p) * memory access + p * page fault cost.
  - Page fault service cost is the sum of
    - Page fault overhead.
    - Swap page out.
    - Swap page in.
    - Restart overhead.
- Example: demand paging performance.
  - Memory access time = 200 ns.
  - Average page fault service time = 8 ms.
  - EAT = (1-p) * 200 ns + p * 8 ms.
  - If one access out of 1000 causes a page fault, then EAT = 8.2 ms.
  - What if we want slowdown by less than 10%? p = 1/400,000.
- Factors lead to misses.
  - Compulsory misses: Pages that have never paged into memory before.
    - Prefetching: Loading them into memory before needed.
    - Need to predict future somehow.
  - Capacity misses: Not enough memory. Must somehow increase size.
    - One option: Increase amount of DRAM.
    - Another option: If multiple processes in memory, adjust percentage of memory allocated to each one.
  - Polycy misses: Caused when pages were in memory, but kicked out prematurely because of the replacement policy.
    - So we need a better replacement policy.

### Page replacement policy

- Find some page in memory, but not really in use, swap it out.
  - Performance: We want an algorithm which will result in minimum number of page faults.
  - Same page may be brought into memory several times.
- Basic page replaement.
  - Find the location of the desired page on disk.
  - Find a free frame: If there is a free frame, use it. If there is no free frame, use a page replacement.
    - It's algorithm to find a victim frame.
  - Swap out: Use modify (dirty) bit to reduce overhead of page transfers. Only modified pages are written to disk.
  - Bring the desired page into the free frame. Update the page and frame tables.
  - The more physical frames you have, the less page faults there will be.
    - Not necessarily true for FIFO (Belady's anomaly). After adding memory. With FIFO, contents can be completely different. In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 pagse.
- Page replacement policies
  - FIFO. Throws out oldest page.
    - Fair. Let every page live in memory for same amount of time.
    - Bad. Throws out heavily used pages instead of infrequently used pages.
  - MIN (minimum) or OPT (optimum). Replace page that won't be used for the longest time.
    - Great but cannot really know future.
    - Makes good comparison case, however.
  - Random. Pick random page for every replacement.
    - Typical solution for TLB's. Simple hardware.
    - Pretty unpredictable. Makes it hard to make realtime guarantees.
- LRU (least recently used)
  - Replace page that hasn't been used for the longest time.
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
  - Use a list to implement LRU. On each use, remove page from list and place at head. LRU page is at tail.
  - Need to know immediately when each page used so that can change position in list.
  - Many instructions for each hardware access.
  - To implement LRU, use a key-value map or bitmap and double linked list. But it is too expensive to implement in reality for many reasons (6 pointer updates when accessing a page).
  - In practice, people approximate LRU.
- LRU approximation.
  - Clock algorithm: Arrange physical pages in circle with single clock hand. Replace an old page, may not the oldest page.
    - Advances only on page fault. Check for pages not used recently. Mark pages as not used recently.
    - What if hand moving slowly: Not many page faults and/or find page quickly.
    - What if hand is moving quickly: Lots of page sfaults and/or lots of reference bits set.
    - One way to view clock algorithm: Crude partitioning of pages into two groups: young and old. Why not partition into more than 2 groups?
  - N-th chance version of clock algorithm.
    - Large N better approx to LRU.

        - Small N is more efficient.
        - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing.
    - Hardware 'use' bit per physical page: Hardware sets use bit on each reference. If use bit isn't set, means not referenced in a long time. Nachos hardware sets use bit in the TLB. You have to copy this back to page table when TLB entry gets replaced.
    - On page fault. Advance clocl hand. Check use bit. If 1 means used recently, clear and leave along. If 0 then select candidate for replacement.
    - Will always find a page or loop forever? Even if all use bits set, will eventually loop around (FIFO).

### SWAP storage

- Swap-space: Virtual memory uses disk space as an extension of main memory.
- Swap-space can carved out of the normal file system, or, more commonly, it can be in a separate disk partition.
- Allocate swap space when process starts; holds text segment (the program) and data segment. Kernel uses swap maps to track swap-space use.
- Issues
    - Initial allocation. Each process needs minimum number of pages.
        - Fixed allocation vs. priority allocation.
    - Where to find frames.
    - Global replacement: Find a frame from all processes.
    - Local replacement: Find only from its own allocated frames.
- Fixed allocation
    - 0 allocation.
    - Equal allocation: For example, if there are 100 frames and 5 processes, give each process 20 frames.
    - Proportional allocation: Allocate according to the size of process.
        - m=64, s1=10, s2=127, a1=10/137x64, a2=127/137x64
- Priority allocation
    - Use a proportional allocation scheme using priorities rather than size.
    - If process Pi generates a page fault
        - select for replacement one of its frames;
        - select for replacement a frame from a process with lower priority number.
- Thrashing: If a process doesn't not have enough pages, the page-fault is very high. This leads to: Low CPU utilization, OS thinks that it needs to increase the degree of multiprogramming, another process added to the system.
    - Thrashing means a process is busy swapping pages in and out.
- Working set.
    - The set of memory locations that a program has referenced in the recent past.
    - Represents data access pattern of program in a time period (locality).
    - As a program executes, it transitions through a sequence of working sets consisting of varying sized subsets of the address space.
    - Great performance if working-sets of all active processes fits into memory.
    - Why does demand paging work? Process migrates from one locality to another. Localities may overlap. Need sufficient memory so that working set fits in memory.
    - When working sets size is larger than total memory size, thrashing occurs.
- Tradeoffs of page size on performance. Impact of page size selection.
    - TLB hit rate: increase.
    - Internal fragmentation: decrease.
    - Total page table size: decrease.
    - I/O overhead (useless I/O). Depend on data access pattern of a program.

### Other related techniques

```
#define num_ints 1000
#define filesize num_ints * sizeof(int)

fd = open(filepath, O_RDWR | O_CREAT | O_TRUNC, 0600);
result = lseek(fd, filesize - 1, SEEK_SET);
result = write(fd, "", 1);
map = mmap(0, filesize, PROT_READ | PROT_WRITE | MAP_SHARED, fd, 0);
for (int i = 1; i <= numints; i++)
    map[i] = 2 * i;
munmap(map, filesize);
close(fd);
```

- Memory mapped files
    - Allow file I/O to be treated as routine memory access by mapping a disk block to a page in memory.
    - Simplifies file access through direct memory access rather than `read()` and `write()` system calls.
    - File access is managed with demand paging.
    - Several processes may map the same file into memory as shared data.
- Review of caching concept
    - Cache: A repository for copies that can be accessed more quickly than the original. Make frequent case fast and infrequent case less dominant.
    - Caching underlies many of the techniques that are used today to make computers fast. Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc.
    - Only good if: Frequent case frequent enough and infrequent case not too expensive.
    - Average access time = (hit rate) x (hit time) + (miss rate) x (miss time).
    - Temporal locality: keep recently accessed data items closer to processor.
    - Spatial locality: data access is contiguous following its layout.
- Zipf distribution
    - Caching behavior of many systems are not well characterized by the working set model.
    - An alternative is the Zipf distribution: Popularity ~ 1/k^c , for k-th most popular item.
    - Caching popular items can yield a very high cache hit ratio. LRU is a good replacement policy that assumes recent accessed items may be accessed again.
    - Cache frequently accessed data in OS.
        - TLB cache. VM. File cache for disk sectors.
    - Cache popular web pages in web servers or in ISP:
        - Popular URLs are accessed again and again.
        - Akamai.com caches popular content in all ISP sites.
    - Cache popular queries in Google.com.
        - Popular queries are entered by many users.
        - Caching results greatly improve search response time.
    - Cache popular nodes in a big social graph.
        - Number of followers in Twitter.
    - Cache information of popular items sold on Amazon.com.
        - Popular items have more chances to be browsed/purchased.

## CPU Scheduling

- Life cycle (states) of a process or thread. Active processes/threads transit from Ready queue to Running to various waiting queues.
- Question: How is the OS to select from each queue. Obvious queue to worry about is ready queue. Others can be scheduled as well, however.
- Scheduling: Deciding whihc processes/threads are given access to resources.
    - Job scheduling, resource scheduling, request scheduling.
    - For example, web server scheduling to handle high traffic.
- Process execution consists of a cycle of CPU execution and I/O wait. Scheduling happens among CPU/IO bursts.
- Process state change vs. CPU scheduling.
    - CPU scheduling decisions may take place when a process:
        - Switches from running to waiting (blocked) state.
        - Switches from running to ready state.
        - Switches from waiting/blocked to ready.
        - Terminates.
    - Preemptive scheduling takes the processor away from one process (job) and give it to another. State change from running to ready.
    - Non-preemptive scheduling. A process runs continuously until it is blocked or terminates.
- Terminalogy
    - Task/Job. Process, thread. User request. E.g., mouse click, web request, shell command, etc.
    - CPU utilization. Keep the CPU as busy as possible.
    - **Throughput**. Number of processes that complete their execution per time unit.
    - **Waiting time**. Amount of time a process has been waiting in the ready queue.
    - **Turnaround time**. Amount of time to execute a particular process.
        - Completeion time - arrival time = waiting time + job size + overhead.
    - **Response time**. Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

### Scheduling algorithm

- Scheduling algorithm optimization criteria.
    - Max CPU utilization. Max throughput. Min turnaround time. Min waiting time. Min response time.
    - Four preemptive or non-preemptive algorithms are considered: FIFO, SJF, priority scheduling, round robin.
- **FIFO**: First-in-first-out, aka first-come-first-served (FCFS)
- **SJF**: Always do the task that has the shortest remaining amount of work to do. Often called shortest remaining time first (SRTF).
    - SJF is optimal. Gives minimum average waiting time for a given set of processes. The difficulty is knowing the length of the next CPU request.
    - Weakness: Starvation. Low priority processes may never execute.
- Predicting length of next CPU burst.
    - Why? SJF scheduling requires size information predict by using the length of previous CPU bursts. Using expontial averaging. New prediction is weighted average of previous prediction and actual time.
    - $t_{n+1} = \alpha t_n + (1-\alpha)t_n$, where
        - $t_n$ is the acutal length of n-th CPU burst.
        - $t_{n+1}$ is the predicted value for the next CPU burst.
        - $0 \leq \alpha \leq 1$.
- **Priority scheduling**.
    - A priority number (integer) is associated with each process.
    - The CPU is allocated to the process with the highest priority (smaller ingeger means higher priority).
    - SJF is a priority scheduling where priority is the predicted next CPU burst time.
    - Problem is starvation. Low priority processes many never execute.
    - Solution: Aging (as time progresses, decreasing the priority of the long-running process). Round robin.
- **Round robin (RR) scheduling**
    - Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds.
    - After this time has elapsed, the process is preempted and added to the end of the ready queue.
    - Performance. Large quantum = FIFO. Small quantum = context switch overhead is too high.
    - Typically, higher average turnaround time than SJF, but better response. Also, no starvation.
    - Compared FIFO: Everybody finishes very late with overhead. Longer turnaround. But faster response. Relatively fair.
- **Overhead during job switches**
    - Process context switch: CPU switch from one process to another process.
    - Code executed in kernel above is overhead. Overhead sets minimum practical switching time. Less overhead with SMT/hyperthreading, but contention for resources instead.
    - Context switchin in Linux: 3-4 $\mu$s (Intel i7, E5).
    - Thread switching only slightly faster than process switching (100 ns).
    - But switching across cores about 2x more expensive than within-core switching.
    - Context switching time increases sharply with the size of the working set, and can increase 100x or more.
    - The working set is the subset of memory used by the process in a time window.
    - Moral: Context switching depends mostly on cache limits and process or thread's hunger for memory.
- **Multi-level feedback queue (MFQ)**
    - Goals: Responsiveness, especially for interactive/high priority jobs. Less overhead. Fairness (among equal priority tasks). No starvation.
    - Not perfect at any of them: FIFO, SJF, RR. MFQ addresses this: used in Linux and Windows.
    - Maintain multiple job queues. Each queue receives a fixed percentage of system resource. A process can move between the various queues, representing priority aging so that high priority jobs will gradually lose their priority.
- MFQ Example
    - Ready queue is partitioned into separate queues: foreground (interactivge) and background (batch).
    - Each queue has its own scheduling algorithm: foreground (RR) and backgorund (FIFO).
    - Each queue gets a certain amount of CPU time which it can schedule amongst its processes, i.e., 80% to foreground in RR, 20% to background in FIFO.

    - Q0: RR with time quantum 8 milliseconds, Q1: RR time quantum 16 milliseconds, Q2: FIFO.
    - Scheduling. A new job enters Q0 which is served FCFS. When it gains CPU, job receives 8 ms. If it does not finish in 8 ms, job is moved to Q1. At Q1 job is again served FCFS and receives 16 ms. If it still doesn't complete, it is preempted and moved to Q2.

### Scheduling examples

- Windows scheduling
    - Real time priority class: static priorities (priorities do not change). Priority values from 16 to 32.
    - Variable class: variable priorities (e.g., 1-16). If a process has used up its quantum, lower its priority. If a process waits for an I/O event, raise its priority.
    - Priority-driven scheduler. For real-time class, do RR within each priority. For variable class, do MFQ.
- Linux scheduling
    - Time-sharing scheduling: Each process has a priority and # of credits.
        - Every clock tick the running process lost a credit. Long running jobs lose credits.
        - When it reached 0, another process with most credit won was chosen.
        - The crediting rule: Credits = credits / 2 + priority.
        - I/O event will raise the priority: fast response time when ready.
    - Real-time scheduling. Soft real-time. Kernel cannot be preempted by user code.
- Thread scheduling
    - Scheduling user-level threads within a process. Known as process-contention scope (PCS).
    - Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
- Pthread scheduling
    - API allows specifying either PCS or SCS during thread creation.
    - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling.
    - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling.
- Multiple-processor scheduling
    - CPU scheduling more complex when multiple CPUs are available.
        - Each processor is self-scheduling. Each has its own private queue of ready processes.
        - Or all processes in common ready queue.
    - Process has affinity for processor on which it is currently running.

### Summary

- **Scheduling**. Selecting a process from the ready queue and allocating the CPU to it.
- **FIFO (FCFS)**
    - Pros: Simple.
    - Cons: Short jobs get stuck behind long ones.
- **RR**
    - Pros: Better for short jobs. Relatively fair.
    - Cons: Weak when jobs are smae legnth. No prioritization. Longer average turnaround. More context switch overhead.
- **SJF**
    - Pros: Optimal (averagfe response time).
    - Cons: Hard to predict future, unfair.
- **MFQ**
    - Fairness while having different priorities. Everybody makes progress.
    - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF.
    - Responsiveness, especially for interactive/high priority jobs.
    - Less context switch overhead with different quantum.

## File Systems

### Files

- **Files**: Contiguous logical address space in a persistent storage (e.g., disk).
- **File structure**: The OS and program decides the structure.
    - None: sequence of words and bytes.
    - Simple record structure: lines, fixed length, variable length.
    - Complex structure: formatted document.
- **Attributes**: Name, identifier, type, location ,size, protection, time, date, user identification, etc.
- **Operations**: Create, open, close, write, read, reposition within file, delete, truncate.
- **Access methods**: Sequential access, direct access.
- **File system abstraction**
    - **Directory**: group of named files or subdirectories. Mapping from file name to file metadata location.
    - **Path**: String that uniquely identifies file or directory.
    - **Links**: Hard links from name to metadata location; soft links from name to alternative name.
    - **Mount**: Mapping from name in one file system to root of another.

### UNIX file system interface

- **UNIX file system API**
    - create, link, unlink, createdir, rmdir.
    - open, close, read, write, seek.
    - fsync: File modifications can be cached, fsync forces modifications to disk (like a memory barrier).
- **Protection**: File owner/creator should be able to control what can be done by whom.
- **Access lists and groups in Linux**
    - **Mode of access**: Read, write execute.
    - **Three classes of uses**: Owner, group, public.
        - Ask manager to create a group (unique name), say `G`, and add some users to the group. For a particular file (say game) or subdirectory, define an appropriate access.
- **Directory structure**: A collection of nodes containing information about all files. Operations performed include search, create, delete, list, rename, traverse.
    - **Name resolution**: The process of converting a logical name into a physical resource (like a file): (1) Traverse succession of directories until reach target file and (2)Global file system may be spread across the network.

```c
// file content: This is a test file
int main() {
    int file = 0; char buffer[15];
    if ((file = open("testfile.txt", O_RDONLY)) < -1) return 1;
    if (read(file, buffer, 14) != 14) return 1;
    printf("%s\n", buffer); // This is a test
    if (lseek(file, 5, SEEK_SET) < 0) return 1;
    if (read(file, buffer, 19) != 14) return 1;
    printf("%s\n", buffer); // is a test file
}
```

### File system structure

- **File system** is a layer of OS that transforms block interface of disks (or other block devices) into files, directories, etc.
- **File system components**
    - **Disk management**. Collectign disk blocks into files.
    - **Naming**. Interface to find files by name, not by blocks.
    - **Protection**. Layers to keep data secure.
    - **Reliability/Durability**. Keeping of files durable despite crashes, media failure, attacks, and etc.
- **User vs. system POV**
    - User's view: Durable data structures.
    - System call interface: Collection of bytes (UNIX).
    - System's view (inside OS): Collection of blocks.
        - A block is a logical transfer unit, while a sector is the physical transfer unit on disk).
        - Block size >= sector size; in UNIX, block size is 4KB.
- **Translating from user to system view**
    - What happens if users says: Give me bytes 2 to 12? (1) Fetch block corresponding to those bytes. (2) Return just the correct portion of the block.
    - What about: Write bytes 2 to 12? (1) Fetch block. (2) Modify portion. (3) Write out block.
    - Everything inside file system is in whole size blocks. E.g., `getc()` and `putc()` buffer something like 4KB, even if interface is one byte at a time.
    - From now on, a file is a collection of blocks.

### File system design

- **Data structures**
    - Directores: File name to file metadata. Store dirs as files.
    - File metadata: How to find file data blocks.
    - Free map: List of free disk blocks.
- How do we organize these data structures? Device has non-uniform performance.
- Design challenges: Index structure, index granularity, free space, locality, reliability.
- **File system workload**. Studying workload characteristics can help feature prioritization or optimization of design. What should be considered? **File sizes**, and **file access patterns**.
    - **Sequential access**: Bytes read in order. Mose of file accesses are of this flavor.
    - **Random access**: Read/write element out of middle of array. Less frequent, but still important. Want this to be fast.
    - **Content-based access**: Many systems don't provide this. Instead, build DBs on top of disk access to index content (requires efficient random access).

### File system implmentation

- **Directories and index structure**. Special root block at a specific location contains the root directory. DIrector structure organizes the files:
    - Given file name, find a file number.
    - Given a file number which contains the file structure info, locate blocks of this file.
- Per-file **file control block (FCB)** contains many details about the file, called **inode** on *nix.
    - A typical FCB include file permissions, dates, owner, group, ACL, size, data blocks of pointers to file data blocks.
- **Layered file system**: APplication programs, logical file system, file-organization module, basic file system, I/O control, and devices.
    - Virtual file systems (VFS) provide an OO way of implementing file systems.
    - VFS allows the same system call interface (the API) to be used for different types of file systems.
    - The API is to the VFS interface, rather than any specific type of file system.

### Directory implementation

- Naive approaches: Linear list, hash table, search tree.
- **All information about a file contained in its file header**. inodes are global resources identified by index (inumber). Once you load the header structure, all blocks of file are locatable.
    - The maximum number of inodes is fixed at file system creation, limiting the maximum number of files the file system can hold.
    - A typical allocation of heuristic for inodes in a file system is one percent of total size.
    - The inode number indexes a table of inodes in a known location on the device.
- **Directory layout**. Directory are stored as a file. Linear search to find filename (for small directories). B-trees are used for large directores.
- **Example: resolve /my/book/count**
    - Read in file header for root `/` (fixed spot on disk).
    - Read in first data block for root `/`; search for `my`.
        - Table of filename/index pairs. Search is done in linear. OK since directories typicall very small.
    - Read in file header for `my`.
    - Read in first data block for `my`; search for `book`.
    - Read in file header for `book`.
    - Read in first data block for `book`; search for `count`.
    - Read in file header for `count`.
- **Current working directory (CWD)**. Per-address-space pointer to a directory (inode) used for resolving file names. Allows user to specify relative filename instead of absolute path (say `CWD=/my/book/` can resolve `count`).
- **Open system call**
    - Resolve file name, finds FCB (inode).
    - Makes entries in per-process and system-wide tables.
    - Return index (called file descriptor or file handle) in open-file table.
- Several pieces of data are needed to manage open files.
    - File pointer: Pointer to last read/write location, per process that has the file open.
    - File-open count: Counter of number of times a file is open, allowing removal of data from open-file table when last processes closes it.
    - Disk location of the file: Cache of data access information.

    - Access permissions: Per-process access mode information.
    - Open file locking is proceded by some systems to mediates access to a file.
- **Read/write system calls**
    - Use file handle (descriptor) to locate inode.
    - Perform appropriate reads or writes.

### Allocation of disk blocks

- An allocation method refers to how disk are allocated for files:
    - Contiguous allocation.
    - Linked allocation.
    - Indexed allocation.
- **Contiguous allocation**. Each file occupies a set of contiguous blocks on the disk.
    - Pros: Simple (only starting location (block#) and length (number of blocks) are required). Fast random access.
    - Cons: Not easy to grow files. Waste in space (e.g., external fragmentation).
- **Linked allocation**. Each file is a linked list of disk blocks. Blocks may scattered anywhere on the disk.
- **Microsoft file allocation table (FAT)**
    - Linked list index structure: Simple, easy to implement. Still widely used.
    - File table: Linear map of all blocks on disk. Each file is a linked list of blocks.
    - Pros: Easy to find free block, to append to a file, to delete a file.
    - Cons: Small file access is slow. Random access is very slow. Fragmentation (File blocks for a given file may be scattered. Files in the same directory may be scattered. Problems becomes worse as disk fills).
- **One-level indexed allocation**. Place all direct data pointers together into the index block. E.g., Nachos file control block has 32 data block pointers, 128 bytes per block.
    - Pros: Support random access. No external fragmentation.
    - Cons: Space overhead (need 1 block for index table).
- **Two-level indexed allocation: single indirection**. Max size is 1k * 1k * 4KB = 4GB.
- **Hybrid multi-level scheme (Unix file system)**. Efficient for small files, but still allow big files. File header contains 13-15 pointers. File header (inode) format: 10-12 direct data pointer (4KB each), 1 indirect block (4MB each), 1 doubly indirect block (4GB each), 1 triple indirect block (4TB each).
- **Free space management**. Block number calculation = number of bits per word * number of 0-value words + offset of first 1 bit.
- **Performation optimization**
    - Disk cache: Separate section of main memory for frequently used blocks.
    - Read-ahead (prefectching): Techniques to optimize sequential access.
    - Improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

## Mass-Storage Systems

- **Magnetic tape**
    - Relatively permanent and holds large quantities of data.
    - Random access ~1000 times slower than disk.
    - Mainly used for backup, storage of infrequently-used data, transfer medium between systems.
    - Typical storage 1.5-20 TB.
    - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT.
- **Disk attachment**
    - Drive attached to computer via I/O bus. USB, SATA (replacing ATA, PATA, EIDE).
    - SCSI: Itself is a bus, up to 16 devices on one cable, SCSI initiator requests operation and SCSI targets perform tasks.
    - FC (fiber channel) is high-speed serial architecture. Can be switched fabric with 24-bit address space. The basis of storage area networs (SANs) in which many hosts attach to many sotarge units. Can be arbitarted loop (FC-AL) of 126 devices.
- **Network-attached storage**
    - Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus).
    - NFS and CIFS are common protocols.
    - Implemented via remote procedure calls (RPCs).
    - New iSCSI protocol uses IP network to carry the SCSI protocol.
- **Storage area network**
    - Special/dedicated network for accessing block level data storage.
    - Multiple hosts attached to multiple storage arrays (flexible).

### Disk

- Drives rotate at 60 to 200 times per second.
- Positioning time is time to move disk arm to desired cylinder (seek time), plus time for desired sector to rotate under the disk head (rotational latency).
- Effective bandwidth: Average data transfer rate during a transfer, that is, the number of bytes divided by transfer time. Data rate incldues positioning overhead.
- Disk latency = seek time + rotation time + transfer time.
    - Seek time: Time to move disk arm over track (1-20ms). Fine-grained position adjustment necessary for head to "settle" head switch time ~ track switch time (on modern disks).
    - Rotation time: Time to wait for disk to rotate under disk head (4-15ms). On average, only need to wait half a rotation.
    - Transfer time: Time to transfer data onto/off of disk. Disk head transfer rate: 50-100MB/s (5-10 usec/sector). Host traansfer rate dependent on I/O connector (USB, SATA, ...).
- Example: 7200 RPM, 54 MB/s, seek time 10.5 ms. How long to complete 500 random disk reads in FIFO order? Each reads one sector (512 bytes).
    - Seek average 10.5 ms.
    - Rotation time = 1/120 (time per rotation) * 0.5 (average rotation) = 4.15 ms.
    - Transfer time: 54 MB/s to transfer 512 bytes per sector = 0.005 ms.
    - In total: 500 * (105 + 4.15 + 0.005) / 1000 = 7.3 seconds.
    - Effective bandwidth: 500 sectors * 512 bytes / 7.3 seconds = 0.034 MB/s. Copying 1 GB of data takes 8.37 hours.
- Same question. How long to complete 500 sequential disk reads?
    - Seek time: 10.5 ms (to reach the right track).
    - Rotation time: 4.15 ms (to reach first sector).
    - Transfer time for all 500 sectors = 500 * 512 / 128 = 2 ms.
    - Total: 10.5 + 4.15 + 2 = 16.7 ms.
    - Effective bandwidth: 500 * 512 / 16.7 = 14.97 MB/s. This is 11.7% of the maximum transfer rate with 250 KB data transferring.

### Disk scheduling

- **Objective**: Given a set of I/O requests. Coordinate disk access of multiple I/O requests for faster performance and reduced seek time.
    - Seek time ~ seek distance.
    - Measured by total head movement in terms of cylinders from one request to another.
- **FCFS scheduling** (first come first serve).
- **SSTF scheduling** (shortest seek time first). Selects the request with the minimum seek time from the current head position.
- **SCAN: elevator algorithm**. Move disk arm in one direction until all requests satisfied, then reverse direction.
- **C-SCAN** (circular SCAN). Provides a more uniform wait time than SCAN by treating cylinders as a circular list. The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

### Solid state disks (SSDs)

- Use NAND multi-level cell (2-bit/cell) flash memory. Non-volatile storage technology. Sector (4 KB page) addressable, but stores 4-64 "pages" per memory block. No moving parts (no rotate/seek motors). Very low power and lightweight
- Transfer time: transfer 4KB page. Limited by controller and disk interface (SATA: 300-600 MB/s).
- Latency = queuing time + controller time + transfering time.
- **SSD architecture: Writes**
    - Writing data is complex (~200µs-1.7ms).
    - Can only write empty pages in a block.
    - Erasing a block takes ~1.5 ms.
    - Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reverses some % of capacity.
    - Data written in 4KB pages, and erased in 256 KB blocks.
    - Controller garbage collects obsolete pages by copying valid pages to new (erased) block.
    - Typical steady state behavior when SSD is almost full. One erase every 64 or 128 writes.
    - Write and erase cycles require high voltage. Damages memory cells, limits SSD lifespan. Controller uses ECC, performs wear leveling. Wear leveling and garbage collection cause data to be rewritten on the SSD.
    - Result is very workload dependent performance.
    - Latency = queuing time + controller time (find free block) + transfering time.
    - Rule of thumb: Wrties 10x more expensive than reads, and writes 10x mroe expensive than writes.
- Pros (vs. HDD): Low latency, high throughput. No moving parts. Read at memory speeds.
- Cons: Small storage, very expensive. Assymetric block write performance: read page erase/write page. Limited drive lifetime.

### Hybird disk drive

- A hybrid disk uses a small SSD as a buffer for a larger drive.
- All dirty blocks can be flushed to the actual hard drive based on: Time, Threshold, Loss of power/computer shutdown.

## Reliable Storage

- **Availability**. The probability that the system can accept and process requests.
    - Often measured in "nines" of probability.
    - Key idea here is independence of failures.
- **Durability**. The ability of a system to recover data despite faults.
    - This idea is fault tolerance applied to data.
        - Mean time before failure (MTBF). Inverse of annual failure rate.
        - Mean time to repair (MTTR) is a basic measure of the maintainability of repairable items. It represents the average time required to repair a failed component or device.
    - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone.
    - How to make file system durabile?
        - Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive. Can allow recovery of data from small media defects.
        - Make sure writes survive in short term. Either abandon delayed writes or use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
        - Make sure that data survives in long term. Need to replicate. Independence of failure.
- **Reliability**. The ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition).
    - Usually stronger than simply availability: means that the system is not only "up", but also working correctly.
    - Includes availability, security, fault tolerance/durability.
    - Must make sure data survives system crashes, disk crashes, etc.

### RAID (redundant array of inexpensive disks)

- Multiple disk drives provide reliability via *redundancy*.
    - Increases the mean time to failure.
    - Improve reliability by storing redundant data.
    - Improve performance with disk striping (use a group of disks as one storage unit).
- RAID is arranged into six different levels.
    - Mirroring (RAID 1) keeps duplicate of each disk.
    - Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability.
    - Block interleaved parity (RAID 4,5,6) uses much less redundan.
- RAID 0: Non-redundant disk array.
    - Files are strped across disks, no redundant info.
    - High read throughput, best write throughput.
    - Any disk failure results in data loss.
    - 1 I/O requests to read a byte in a disk block.
    - 2 I/O requests to write a byte in a block. (Read block, modify, and write back.)
- RAID 1: Mirrored disks.
    - Data is written to two places. On failure, just use surviving disk and easy to rebuild.

    - On read, choose fastest to read. 2x performance.
    - Write performance is same as single drive.
    - Expensive (high space overhead).
    - 1 I/O requests to read a byte in a disk block.
    - 3 I/O requests to write a byte in a disk block.
- RAID 0+1: Stripe on a set of disks. Then mirror of data blocks is striped on the second set.
- RAID 1+0: Pair mirrors first. Then stripe on a set of paired mirrors. Better reliability than RAID 0+1.
- RAID 2: Memory-style error-correcting codes.
- RAID 3: Bit-interleaved parity.
- RAID 4: Block-interleaved parity.
- RAID 5: Block-interleaved distributed parity, with redundancy to recover from a single disk failure.
    - Files are striped as blocks. Blocks are distributed among disks. Parity blocks are added.
    - 1 I/O requests to read a byte in a disk block.
    - 4 I/O requests to write a byte in a block. (Read old data block, read old parity block, write new data block, write new parity block = old data XOR old parity XOR new data).
- RAID 6: RAID 5 with extra redundancy to recover from two disk failures.

### More reliable file systems

- **Carefully order operation sequence**
    - Read data structures to see if there were any operations in progress. Clean up/finish as needed.
    - FAT and FFS (fsck) to protect filesystem structure/metadata.
    - Many app-level recovery schemes (e.g., MS Word, emacs autosaves).
- **Copy on write file layout**
    - To update file system, write a new version of the file system containing the update.
    - NetApp's Write Anywhere File Layout (WAFL).
    - ZFS (Sun/Oracle) and OpenZFS.
- **Transactional file systems**. Better reliability through use of log.
    - Applies updates to system metadata using transactions committed once it is written to the disk log.
    - Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional.
    - File system may not be updated immediately, data preserved in the log.
    - Example: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4.

### Transactional file systems

- Typical structure
    - Begin a transation: Get transaction ID.
    - Do a bunch of updates. If any fail along the way, rollback. Or, if any conflicts with other transactions, rollback.
    - Commit the transaction.
- An *atomic sequence* of actions on a storage system (or database).
- It extends concept of atomic update from memory to stable storage.
    - Atomically update multiple persistent data structures.
    - That takes it from one *consistent state* to another.
- The ACID properties of transactions.
    - **Atomicity**. Operations appear to happen as a group, or not at all (at logical level). At physical level, only single disk/flash write is atomic.
    - **Consistency**. Transactions maintain data integrity, e.g., file size cannot be negative. If the DB starts out consistent, it ends up consistent at the end of transaction.
    - **Isolation**. Execution of one transaction is isolated from that of all others; no problems from concurrency.
    - **Durability**. If a transaction commits, its effects persist despite crashes.
- **Logging file systems**
    - Full logging file system: All updates to disk are done in transcations.
    - Instead of modifying data strucytures on disk directly, write changes to a journal/log.
        - Intention list: set of changes we intend to make.
        - Log/journal is append-only.
    - Once changes are on log, safe to apply changes to data structures on disk.
        - Recovery can read log to see what changes were intended.
    - Once chances are copied, safe to remove log.
- **Redo logging**
    - Prepare: Write all changes (in transaction) to log.
    - Commit: Single disk write to make transaction durable.
    - Redo: Copy changes to disk.
    - Garbage collection: Reclaim space in log.
    - Recovery: Read log; redo any operations for committed transactions; garbage collect log.
- **Example: Createing a file**
    - Find free data blocks. Find free inode entry. Find directory entry insertion point.
    - [log] Write map (i.e., mark used). [log] Write inode entry to point to block(s). [log] Write dir entry to point to indoe.
    - Crash during logging: Scan the log. Detect transaction start with no commit. Discard log entries. Disk remains unchanged.
    - Recovery after commit: Scan log, find start. Find matching commit. Redo it as usual.

## Cloud Computing with KV Storages and Distributed File Systems

### Cluster computing

- Motivations: Large-scale data processing on clusters.
- Cost-efficiency: Commodity nodes/network. Automatic fault-tolerance. Easy to use.
- Functions: Automatic parallelization and distribution. Fault-tolerance.
- Typical cluster: 40 nodes/rack, 1k-4k nodes in cluster. 1Gbps bandwidth in rack, 8Gbps out of rack. Each node has 8-16 cores, 32GB RAM, 8x1.5TB disks.
- Why colud computing?
    - Cloud refers to large Internet services running on many machines.
    - Cloud computing refers to services by these companies that let external customers rent cycles.
    - Attractive features: Scalability, elastic computing, ease of use.

### Key value storage

- Handle huge volumes of data. Store (key, value) pair. Used sometimes as a simpler but more scalable database.
- Simple interface: `put(key, value)`, `value = get(key)`.
- The KV-tables used for column-based storage, as opposed to row-based storage typical in older DBMS.
- Example: Amazon DynamoDB/S3, BigTable/HBase/Hypertable, Cassandra, Memcached, BitTorrent distributed file location, Redis, Oracle, etc.
- Key value store is also called distributed hash tables (DHT).
    - Main idea: Partition set of key-values across many machines.
    - Challenges: Fault tolerance, scalability, consistency, heterogeneity.
- Dictonary-based architecture: Have a node maintain the mapping between keys and machines (nodes) that store the values associated with the keys.
    - Recursive query: Having the master relay the requests.
        - Pros: Faster, as typically master/directory closer to nodes. Easier to maintain consistency, as master/directory can serialize puts and gets.
        - Cons: Scalability bottlenect, as all values go through master.
    - Iterative query: Return node to requester and let requester contact node.
        - Pros: Scalability.
        - Cons: Slower, harder to enfore data consistency.
- **Fault tolerance**: Replicate value on several nodes. Usually place replicas on different racks in a datacenter to guard against rack failures.
- **Scalability**
    - More storage: Use more nodes.
    - More requests: Can serve requests from all nodes which a value is stored in parallel. Master can replicate a popular value on more nodes.
    - Master/directory scalability: Replicate it. Partition it, so different keys are served by different masters/directories.
    - Load balacning
        - Directory keeps track of the storage availability at each node. Preferentially insert new values on nodes with more storage available.
        - When a new node is added. Cannot insert only new values on new node. Move values from heavy loaded nodes to the new node.
        - When a node fails. Need to replicate values from fail node to other nodes.
- **Consistency**
    - Need to make sure that a value is replicated correctly. In general, slow puts and faster gets.
    - If concurrent updates, may need to make sure that updates happen in the same order to avoid inconsistent write/write.
    - Read cannot guaranteed to return value of lastest write. Can happen if the master processes requests in different threads. Inconsistent write/read.
    - **Atomic consistency** (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image). Think one updated at a time. Transactions
    - **Eventual consistency**: Given enough time all updates will propagate through the system. One of the weakest form of consistency; used by many systems in practice. Must eventually converge on single key/value.
    - And more: Causual consistency, sequential consistency, strong consistency, etc.
- **Quorum consensus**
    - Improve `put()` and `get()` operation performance.
    - Define a replica set of size N. `put()` waits for acks from at least W replicas. The writer returns after it hears from these replicas. `get()` waits for responses from at least R replicas. Make sure W+R>N.
    - There is at least one node that contains the update.

### Distributed file systems

- The interface is the same as a single-machine file system.
- Distribute file data to a number of machines. Support replication. Support concurrent data access.
- Google file system and HDFS (Hadoop). Optimized for batch processing. Provides redundant storage of massive amounts of data on cheap and unreliable computers.
- **HDFS API**
    - Copy data from the local to HDFS: `copyFromLocal src dest`.
    - Copy data from HDFS to local: `copyToLocal src dest`.
- **Assumptions of GFS/HDFS**
    - High component failure rates. Inexpensive commodity components fail all the time.
    - Modest number of huge files. Just a few million. Each is 100 MB or larger.
    - Files are write-once, mostly appended to. Perhaps concurrently.
    - Large streaming reads.
    - High sustained through favored over low latency.
- **HDFS architecture**
    - Files split into 64MB blocks.
    - Blocks are replicated across several datanodes (default 3) as slaves.
    - Namenode stores metadata (file names, locations, etc) as a master.
    - Files are appended-only. Optimized for large files, sequential reads.
        - Read use any copy, write append to all replicas.
    - Namenode: Maps a file to a fileid and list of block ids and data nodes.
    - Datanode: Maps a blockid to a physical location on disk.
    - Secondary namenode: Backup. Periodic merge of transaction log.
- **Namenode metadata**
    - The entire metadata is in memory. No demand paging of meta-data.
    - Types of metadata: List of files/blocks for each file/datanodes for each block. File attributes.
    - A transaction log: Records file creations, file deletions, etc.
- **Datanode**
    - A block server: Stores data in local file system (e.g., ext3). Stores metadata of a block (e.g., CRC). Serves data and metadata to clients.
    - Block report: Periodically sends a report of all existing blocks to the name node.
    - Facilitates pipelining of data: Forwards data to other specified data nodes.
    - Block placement current strategy: One replica on local node, second on a remote rack, third on the same remote rack. Additional ones are placed randomly.
- **Datanode failure detecture with heartbeat**
    - A network partition can cause a subset of Datanodes to lose connectivity with the Namenode.
    - Namenode detects this condition by the absence of a Heartbeat message.
    - Namenode marks Datanodes without Hearbeat and does not send any IO requests to them.
    - Any data registered to the failed Datanode is not available to the HDFS.
    - Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value.
- **Data pipelining during data block write**
    - Client retrieves a list of DataNodes on which to place replicas of a block.
    - Client writes block to the first DataNode.
    - The first DataNode forwards the data to the next DataNode in the Pipeline.
    - When all replicas are written, the Client moves on to write the next block in file.

- **Ensuring data correctness during reading**
  - Use checksums (e.g., CRC32) to validate data.
  - File creation: Client computes checksum per 512 bytes. Datanode stores the checksum.
  - File accesS: Client retrieves the data and checksum from datanode. If validation fails, client tries other replicas.
- **HDFS properties**
  - HDPS provides a write-once-read-many, append-only access model for data.
  - HDFS is optimized for sequential reads of large files with large blocks (e.g. 64MB)
  - HDFS maintains multiple copies of the data for fault tolerance.
  - HDFS is designed for high-throughput, rather than low-latency.
  - Hadoop jobs (e.g. MapReduce) tend to execute over several minutes and hours.

# Problem Set

## Exercise 1

**Q1** Which of the following services is considered as a common service provided by OS?

1. Compiler
2. User graphical interface
3. **Process and memory management**

**Q2** Which of the following instructions should be privileged and executed in a kernel mode?

1. Set value of application timer based on CPU clock value.
2. Read the clock.
3. Issue a trap signal instruction.
4. Access I/O device.

**Q3** What is the purpose of system calls?

1. They are functions which can be executed in an OS.
2. They allow user-level processes to request services of the operating system.
3. They provide the multi-threading or processing capability.

**Q4** When a process creates a new process using the `fork()` operation, which of the following state is shared between the parent process and the child process?

1. Stack
2. Heap
3. **Shared memory segments**. Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

**Q5** Which of the following statements is true on user-level threads and kernel-level threads?

1. User-level threads and kernel threads are both aware by the kernel.
2. **User threads are scheduled by the thread library and the OS kernel schedules kernel threads.**
3. Kernel or user threads are associated with only one process.

**Q6** Which of the following statements is true on context-switch among threads?

1. **Context switching between threads require saving the value of the CPU registers from the thread being switched out.**
2. Context switching between kernel threads typically do not require saving the value of the CPU registers from the thread being switched out.
3. Context switch restores the CPU registers of the new thread being scheduled if this new thread is a user thread.

**Q7** Trace this program without actually running in a machine.

```
void main() {
    char *cmd[] = {“echo”, “hello”,0};
    int pid, x = 10, y;
    pid = fork();
    if (pid == 0) {
        y=1; x=x+y;
        printf("x is %d. y is %d \n", x,y);
        execvp(cmd[0], cmd);
        printf("x is %d. y is %d \n", x,y);
    } else {
        y=2; x=x+y;
        wait(NULL);
        printf("x is %d. y is %d \n", x,y);
    }
}
```

- Output: `x = 11, y = 1; hello; x = 13, y = 2.`
- The child process's x value is not shared with parent. Thus the modification is not reflected in the parent process.
- Function excevp() loads a new program and thus wipes out the parent's code. But if for any reason, loading this program (shell) fails, the print statement (x=11,y=1) does appear.
- The order of printing is fixed because the parent process waits the child process.

**Q8** How many times does this program print "hello"? (6 times)

```
void main() { fork(); printf("hello\n"); fork(); printf("hello\n"); }
```

**Q9** Given this program started from empty file tmpfile1

```
int main() {
    int pid, fd; char *s1;
    pid = fork();
    fd = open("tmpfile1", O_WRONLY|O_CREAT, 0666);
    if (pid > 0) {
      s1 = "Parent";
    } else {
      sleep(1);
      s1 = "Child";
    }
    write(fd, s1, strlen(s1));
    close(fd);
}
```

Output is `childt`. The child process writes the same file from position 0 after the parent writes from offset position 0 also.

**Q10** Given this program started from empty file tmpfile1

```
int main() {
    int pid, fd; char *s1;
    fd = open("tmpfile1", O_WRONLY| O_CREAT, 0666);
    pid = fork();
    if (pid > 0) {
        s1 = "Parent";
    } else {
        sleep(1);
        s1 = "Child";
    }
    write(fd, s1, strlen(s1)); close(fd);
}
```

Output is `ParentChild`. The child process shares the file descriptors of the parent and thus shares the file seek pointer (offset) also.

## Exercise 2

**Q1** Choose one which does not answer the question on "what is a race condition"?

1. When the behavior of a program relies on the interleaving of operations of different threads.
2. Two or more processes (threads) are reading and writing on shared data and the final result depends on who runs precisely & when.
3. **Two or more processes (threads) are reading and writing on shared data and the final result depends on who runs precisely & when.**

**Q2** A semaphore puts a thread to sleep

1. If it increments the semaphore's value above 0.
2. **If it tries to decrement the semaphore's value below 0.**
3. Until another thread issues a signal to notify this semaphore.
4. Until the semaphore's value reaches a certain number.

**Q3** Given three statements

1. F. A semaphore can be implemented using conditional variables with locks.
2. T. A condition variable can be implemented using semaphores.
3. T. A lock can be implemented using a semaphore.

**Q4** Which statement is false about starvation and deadlock

1. Starvation implies that a thread cannot make progress because other threads are using resources it needs.
2. Starvation can be recovered, for example when the other processes finish.
3. Deadlock is a circular wait without preemption that can never be recovered from.
4. **Starvation is a deadlock.** (P.S. But deadlock is a starvation)

**Q5** Trace the following pthreads C program running on a time sharing OS in dynamic context switching is possible, select one answer on the possible output of x.

```
int x=0;
pthread_mutex_t Lock;
void * tcode(void *arg) {
    pthread_mutex_lock(&Lock);
    x++;
    pthread_mutex_unlock(&Lock);
}

void main() {
    pthread_t t1,t2;
    pthread_mutex_init(&Lock,NULL);
    pthread_create(&t1, NULL, tcode, NULL);
    pthread_create(&t2, NULL, tcode, NULL);
    x++;
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("x: %d \n",x);
}
```

1. 0, 1, 2, 3
2. **1, 2, 3**
3. 2, 3
4. 3

The two threads that perform x++ can make the x value increase from 0 to 2. The main program does x++ also, which is not synchronized and can create a race condition. The main thread can either get x as 0, 1, 2 to increase, which can make the final x value as 1,2, or 3.

**Q6** The following C code is a simplified version of lock implementation using a Pthread semaphore..

```
typedef struct locks {
    sem_t sem;
    pthread_t thread;
} lock_t;

int mutex_init(lock_t *lockp) {
    //return 0 when successful
    lockp->thread= pthread_self();
    return sem_init(&lockp->sem,0,1); // (1)
}

int mutex_lock(lock_t *lockp) {
    // return 0 when successful
    if (pthread_equal(lockp->thread, pthread_self()))
        return sem_wait( &lockp->sem); // (2)
    return -1;
}

int mutex_unlock(lock_t *lockp) {
    // return 0 when successful
    if (pthread_equal(lockp->thread, pthread_self()))
        return sem_post(&lockp->sem); // (3)
    return -1;
}
```

**Q7** The following program manages a set of free memory blocks used by multiple threads. Each thread calls `allocate()` to get a free block to use and calls `free()` to release a specific block so others can use. The program maintains an array `available[NB]` whose elements are non-zero if the corresponding blocks are available (`NB` is a constant indicating the total memory blocks available initially).

```
int available[NB];
int allocate() { /* Returns index of available free blocks. */
    while(1)
        for (int i=0; i < NB; i++)
            if (available [i] != 0) { available[i] = 0; return i; }
}
free (int i) { available[i] = 1; }
```

Modify the above code using 1) locks and condition variables so that the multiple threads are well symphonized in accessing the critical section and busy waiting is minimized. 2) using semaphores only. You can express your solution using either Pthreads or Nachos synchronization primitives.

Lock and condition variable solution

```
// Next condition indicates there is no memory available and threads are
Condition* noMemory = new Condition("noMemory");
// Next lock is used for mutual exclusion in checking the available array
Lock *mutex = new Lock("mutex");

int allocate() {
    mutex->Acquire();
    while(1) {
      for (int i = 0; i < NB; i++)
          if (available [i] != 0) { available[i] = 0; mutex->Release(); r
      noMemory->Wait(mutex);
    }
}
int free(int i) {
    mutex->Acquire(); available[i] = 1; noMemory->Signal(mutex); mutex->R
}
```

Semaphore solution

```
// next semaphore indicates if there are free memory blocks available.
// Initially there are NB blocks available
Semaphore *freeBlocks = new Semaphore("Free Blocks", NB);
// Next semaphore is used for mutex exclusion in accessing available arra
Semaphore *mutex = new Semaphore("mutex", 1);

int allocate() {
    while(1) {
        freeBlocks->P();
        mutex->P();
        for (int i = 0; i < NB; i++)
            if (available [i] != 0) { available[i] = 0; mutex->V(); retur
        mutex->V();
    }
}
void free(int i) {
    mutex->P(); available[i] = 1; freeBlocks->V(); mutex->V();
}
```

### Exercise 3

**Q2** What is the possible output of running this nachos program `ThreadTest()`? Give an explanation without actually running this code on a machine.

```
int x = 10;
void SimpleThread(int y) {
    printf(" Before yield, Thread %d: x is %d.\n", y, x);
    currentThread->Yield();
    x = x + y;
    printf(" After yield, Thread %d: x is %d. \n", y, x);
}
void ThreadTest() {
    Thread *t = new Thread("forked thread");
    t->Fork(SimpleThread, 1);
    SimpleThread(2);
}
```

- **Before yield, Thread 2: x is 10. Before yield, Thread 1: x is 10. After yield, Thread 2: x is 12. After yield, Thread 1: x is 13.**
- Before yield, Thread 1: x is 10. Before yield, Thread 2: x is 10. After yield, Thread 1: x is 11. After yield, Thread 2: x is 13.

The parent thread places the child thread in the ready queue, and executes `SimpleTread(2)` first and prints, then it yields. The child thread starts now and prints one sentence, then yields.

**Q4** Which of the following steps is not necessary to run a program.

1. Obtain CPU cycle and allocate memory.
2. Load code/data into memory. Setup stack/heap.
3. **Copy the allocated in-memory content to disk in case the program is being swapped out.**
4. Load starting address and begin execution.
5. Provide the service with address translation and system calls, and control the execution.

**Q5** Which statement is not true regarding difference between logical and physical addresses.

1. A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space.
2. **A physical address refers to an address of a data item used by a program.**
3. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU).
4. Physical addresses are generated by the MMU.

**Q6** Consider a logical address space of 64 pages of 1KB each, mapped onto a physical memory of 32 pages. How many bits are there in the logical address? How many bits are there in the physical address?

1. Logical address: 64 bits. Physical address: 32 bits
2. **Logical address: 16 bits. Physical address: 15 bits**
3. Logical address: 16 bits. Physical address: 16 bits
4. Logical address: 15 bits. Physical address: 15 bits

**Q7** Given a one-level paging scheme for memory address mapping, if TLB access takes 10 nanosecond and each physical memory access takes 100 nanoseconds, what will be the desired TLB hit ratio to have the effective memory access time within 120 nanoseconds to complete both logical address translation and physical memory data fetching?

Solve $10 + (1 - x) * 100 + 100 = 120$ for $x = 0.9$.

**Q8** Given a one-level paging scheme where physical memory is divided into a set of pages with uniform size 8K bytes and each page table entry uses 4 bytes, what is the maximum size of a logical memory space for each process with this paging scheme? Explain your reason.

The one-level page table has to fit into one page and thus at most 8K bytes. Since each entry uses 4 bytes, there are 8K/4 = 2K entries in a page table. Thus there are total of 2K pages, the maximum size of logical memory space is 2K*8KB = 16MB.

**Q9** Design a 2-level paging scheme for a 38-bit logical space and illustrate the address translation steps. Each physical page is of size 16K bytes and each page table entry uses 4 bytes. Each 38-bit logical address is translated into a 38-bit physical address.

- The scheme is 2-level paging and 38-bit will be split to 12, 12, 14.
- Physical page size is 16K bytes, and bits for a page use log(16K) = 14 bits.
- Each physical page can host 16K/4 =4K entries. Thus a page table using one physical page can map at most 4K x 16K=64MB physical space. A two-level table can map at most 64MB*4K=256GB space which is 2^38.
- The outer table has 4K entries, and log(4K)=12. That needs 12 bits to identify a particular entry. Thus the outer table part of the logical address needs 12 bits.
- Consequently, the inner table part of logical address needs 12 bits.

**Q10** Design a 3-level paging scheme for a 64-bit logical space. Each physical page is of size 4K bytes and each page table entry uses 4 bytes.

(10, 10, 10, 12)

**Q11** What is the maximum size of logical and physical spaces that can be mapped by the 3-level paging scheme in Question 10?

- Logical: 4TB.
- Physical: 16TB = 2^32 x 2^12 = 2^34.

**Q12** The following C program access a C array with a million of integer elements, running on a Linux machine. How does increasing of the page size affect the TLB miss rate? Explain the reason.

```
int i, sum = 0;
for (i = 0; i < 10000000; i++)
    sum += a[i];
```

1. **The increasing of the page size can decrease TLB miss rate**
2. The decreasing of the page size can decrease TLB miss rate
3. The increasing of the page size does not affect TLB miss rate
4. The decreasing of the page size does not affect TLB miss rate

For example, the typical size of Linux OS page is 4KB and each integer in C takes 4 bytes. Thus each page

can host 1K integers. This program accesses the array a[] in a consecutive way. Thus when TLB misses the first element of a page, it will hit the rest of integer numbers in this page. The TLB miss rate is 1 out of 1K. When increasing the page size to 8K, the TLB miss rate decreases to 1 out of 2K.

### Exercise 4

**Question 1** Given the following C code, assume x is allocated to a register. Initially the binary code (text) and stack segments of this program is in memory, but others segments are not in memory. All OS kernel pages are in memory.

```
int x=1; // Line 1
x = open("tmpfile1", O_WRONLY, 0666); //Line 2
close(x); // Line 3
```

The execution of the following lines in above code triggers a page fault:

1. Line 1, Line 2, Line 3
2. Line 2, Line 3
3. **Line 2** accesses "tmpfile1" which sits in the initialized data segment which is not in memory.
4. Line 3

**Question 2** Which statement is not true when a page fault occurs?

1. The process that causes this page fault may enter a sleep mode.
2. A free frame is located and I/O is requested to read the needed page into the free frame.
3. Upon completion of page fault handling, the page table of the process that triggers page fault is modified. The corresponding instruction is restarted again.
4. **The process of the victim page will be locked until this page is available again in memory**.

**Question 3** Which of the following data access patterns are often "good" for a demand-paged environment in terms of exploiting data locality for good time efficiency? Which are often bad?

1. Stack-based assignments (Good: Stack with push and pop calls. Access of a consecutive part of a stack exhibits a good spatial locality.)
2. Sequential search (Good: Scan a list of elements sequentially and one can use an array to implement. Thus it tends to have a good spatial locality.)
3. Binary search (Bad: Binary search using a tree accesses a set of data objects scattered in memory and there is no access locality. Binary search of a big sorted array jumps from one memory address to another, which has no locality.)
4. Object traversal with many pointer indirections (e.g., linked list or graph traversal) (Bad: Jumping around objects does not exhibit spatial or temporal locality normally.)

Explanation: Physical memory is a cache for virtual pages. If a program accesses data has some temporal or spatial locality, such a program has a better performance (content of a virtual page loaded to memory can be used again and again).

**Question 4** Consider a demand-paging system that costs about 10,000 microseconds to deal with a page fault. Addresses are translated through a TLB and a one-level page table in the main memory. The cost for accessing TLB is considered as 0 and cost of each memory access is 0.1 microsecond. Assume that 99 percent of the accesses is in the TLB and virtual pages translated by TLB are always in memory. For the remaining 1% of the accesses, address translation is completed through the page table, but virtual pages may not reside in memory. What is the effective memory access time when the page fault rate is 0.01%.

1. 1 ms
2. **1.101 ms**
3. 1.2 ms

If there is no page fault, average memory access time in microseconds is $0.99(0 + 0.1) + 0.01(0.1 + 0.1) = 0.101$. Effective memory access when possibility of pages fault is considered: $0.99(0.101) + 0.01(0.101 + 10000) = 1.101$.

**Question 5** Consider the following page-replacement algorithms. Rank these algorithms from the best to the worst according to their page-fault rate: LRU replacement, FIFO replacement, Optimal (MIN) replacement, Second-chance replacement.

1. **MIN, LRU, Second-chance, FIFO**
2. MIN, Second-chance, LRU, FIFO
3. MIN, FIFO, LRU, Second-chance
4. MIN, LRU, FIFO, Second-chance

**Question 6** Given 3 empty physical pages as the total memory available, an application access memory in the following page reference sequence (x,y,z,w,y,u,y). Each letter represents a distinct virtual memory page. What is the number of page faults with the LRU, second-chance, and FIFO replacement algorithms respectively?

1. **5, 5, 6**
2. 5, 6, 6
3. 5, 5, 7
4. 6, 6, 6

- LRU: xF, yF, zF, wF (replace x), y, uF (replace z), y
- SC: xF, yF, zF, wF, y, uF, y
- FIFO: xF, yF, zF, wF (replace x), y, uF (replace y), yF (replace z)

**Question 7** Given the following C program which access a 2D integer array `a[1024][1024]`. A demand-paging memory is divided into a set of uniform 1K byte pages. Each integer uses 4 bytes. Assume that the binary instructions for the following program are always in separate memory space. The LRU page replacement algorithm is used to manage a 512KB memory that stores the array. Initially data array `a[][]` is not in memory.

```
for (i = 0; i < 1024; i++) for (j = 0; j < 1024; j++) a[j][i] = 100;
```

What is the number of page faults in running this program?

1. 1024
2. **1024 * 1024**
3. 0

The C program stores this array in a row-wise format. Each row needs 1024*4/1K = 4 memory pages. 512KB memory can hold 512 pages. The code accesses column 1, requiring the loading of 1K pages with 1K page faults. For column 2, old data loaded is swapped out, and thus it takes another 1K page faults. In summary, 1024x1024 page faults occur.

**Question 8** For 7, if the memory increases to 1MB, what is the number of page faults in running this program?

1. 1024
2. 1024 * 1024
3. **4096**
4. 0

If memory is 1MB, then memory can hold 1K pages. Each row of the matrix requires 4 pages as it has size 4KB. After column 1 access, data in memory can still be used for column 2, column 3, ..., up to column 256. Then the miss starts again. Thus we ca reason that accessing each row generates 4 page faults. The total number of page faults will be 1024*4= 4K.

**Question 9** Which statement is false on OS scheduling?

1. Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.
2. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.
3. **Round robin scheduling improves job response time, and average turn-around time.**
4. Different time-quantum sizes are used at different queues at a multilevel feedback queuing system.

**Question 10** Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed in the burst time column. .Assume the context switch cost = 0.

- P1: Arrival time = 0, Burst time = 8
- P2: Arrival time = 1, Burst time = 4
- P3: Arrival time = 2, Burst time = 1

What is the average turnaround time for these processes with the FIFO non-preemptive, SJF non-preemptive, RR preemptive scheduling algorithm with quantum 1?

1. **10, 9, 7.33**
2. 11, 10, 7
3. 10, 6, 6

- FIFO: $(8 - 0 + 12 - 1 + 13 - 2)/3$
- SJF: $(8 - 0 + 9 - 2 + 13 - 1)/3$
- RR: $(13 - 0 + 9 - 1 + 3 - 2)/3$

Remember that turnaround time is finishing time minus arrival time, so you have to subtract the arrival times to compute the turnaround times. FIFO's turnaround time is 11 if you forget to subtract arrival time. Another mistake is that SJF schedules all tasks following the job length without considering their arrival time. SJF (P3, P2, P1). Turnaround time would become $(3 - 2 + 5 - 1 + 13 - 0)/3 = 6$ which is wrong.

### Exercise 5

**Q1** Select a statement which is not true

1. The `open()` operation informs the system that the named file is about to become active.
2. The `close()` operation informs the system that the named file is no longer in active use by the user who issued the close operation.
3. **The OS always have file descriptors available for new files to be opened even other programs do not close files.**
4. The files may not be saved properly if a program does not close all files written while exiting abnormally as OS often buffers writing before writing data out to disk.

**Q2** Given two applications: 1) Print the entire content of a file; 2) Access a record in a file given its index. The position of this record in this file is indexed. They should be accessed:

1. **sequentially, randomly**
2. randomly, sequentially
3. both sequentially
4. both randomly

**Q3** Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file. How would you specify this protection scheme in UNIX/Linux?

1. Specify 10 users as non-accessible.
2. **Put these 4990 users in one group and set the group access accordingly.**
3. Give a soft link of this file to these 4990 users so they are able to access while other 10 users cannot access as they do not know the file name.
4. Not possible.

**Q4** We use indexed allocation to store a file on a disk and we assume that each disk file block is of size 16KB. Each data pointer entry in an i-node takes 4 bytes. Answer the following questions with an explanation.

What is the maximum size of a file with one-level direct mapping (one index block containing a set of direct pointers to data blocks) can map?

1. 16 KB
2. **64 MB**
3. 256 MB

**Q5** For Q4, What is the minimum levels of index indirection required to map a file of 16 gigabytes?

1. One-level direct mapping
2. **Two-level single indirection**
3. Three-level double indirection

$16K/4B = 4K$ entries per index block. The maximum size of a file with single indirect block =

$4K * 16KB = 64MB$. Maximum size for two levels: $4K * 4K * 16K = 256GB$. 16GB would require 2 levels.

**Q6** Given an i-node design as UNIX file control block structure, what is the least number of the set of disk blocks that must be read into memory in order to access a UNIX file "/cs170/final.txt"?

1. 1
2. 2
3. 4
4. **6**

- Read in file header for root / (fixed spot on disk).
- Read in first data block for root / as its directory content and search cs170.
- Read in file header for directory "cs170"
- Read in first data block for directory "cs170"; search for "final.txt"
- Read in the file header i-node for "final.txt"
- Read in first data block for "final.txt"

**Q7** Suppose this disk drive spins at 12000 RPM, and has a sector size of 512 bytes, and holds 50 sectors per track. What is the maximum transfer rate of this driver accomplishable without seek overhead.

1. **5 MB/s**
2. 6 MB/s
3. 12 MB/s

**Q8** For Q7, Assume average seek time for the above drive is 7.3 milliseconds, what is the effective transfer rate for random-access of two consecutive sectors on a disk track? (Namely find the position first, then transfer two sectors together).

1. **0.1 MB/s**
2. 0.133 MB/s
3. 5 MB/s
4. 6 MB/s

- Average rotational cost is time to travel half track: 1/200 * 0.5 = 2.5ms.
- Transfer time = 7.3ms to seek + 0.2ms to read = 10 ms.
- Effective transferring rate: 1KB/0.01s = 0.1MB/s.

**Q9** Suppose that a disk drive has 200 cylinders, numbered from 0 to 199. The drive is currently serving a request at cylinder 120, and the previous request was at cylinder 100. The queue of pending requests, in FIFO order is: 80, 190, 40, 110. Illustrate the disk arm movement using the FCFS and SCAN algorithms. Compute the total movement in cylinders for each algorithm.

1. 220, 370
2. **370, 220**
3. 388, 189
4. 189, 388

- FCFS: 120, 80, 190, 40, 110
- SCAN: 120, 190, 110, 80, 40

**Q10** Assume the annual failure rate of a disk drive is 2% and each server machine has 5 disk drives installed. Given 1000 servers installed in a computer cluster, how often does one of their disk drives fail?

1. Once a day
2. Once a week
3. Once a month
4. **Twice a week**

**Q11** Which statement is false?

1. RAID 0 improves access bandwidth (parallel read or write), but there is no extra redundancy added to deal with a disk failure.
2. RAID 1 improves reliability and can tolerate one disk fail, and it doubles read throughput. But 50% of disk space is wasted.
3. **RAID 5 improves access bandwidth while it can tolerate up to 5 disk failures.**
4. RAID 1+0 improves access bandwidth while it can tolerate one disk failure.

**Q12** Given one data block contains content in binary value 0011, and another block 1011. What is the binary value of the parity block for these two blocks? If the parity block of two blocks A and B is 0011 and block A is 1011, B is lost during a failure. What is B in the binary value that can be recover from the parity block and block A?

1. 0111, 0111
2. **1000, 1000**
3. 0111, 1000
4. 1000, 01111

**Q13** Considering A and B with initial value 1. Transaction T is: A=3; B=4; commit. Select one of following transaction processing that violates ACID properties.

1. **T crashes after A=3. The final state after recovery is A=3, B=1.**
2. T crashes after A=3. The final state after recovery is A=1, B=1.
3. T execution is successful. The final state is A=3, B=4.

**Q14** Answer true or false for each of the following statements

- **T**. For a key-vale store with multiple replicas, using quorum consensus a writer can return before the data has been written to all replicas.
- **T**. The directory server as master can be a bottleneck point for a key-value store.

**Q15** Answer true or false for each of the following statements

- **F**. Hadoop distributed file system allocates 3 replicas by default and accesses a replica for reading randomly.
- **T**. Hadoop distributed file system is optimized for handling large files with large file blocks and heavy sequential read workload.

## F12 midterm

**Q1.1** What are the key OS actions taken during context switching between two processes? What are their process states before and after switching?

- Suspend the current process.
- Save current process control information in a process control block (PCB).
- Restore the process control information for the new process from its PCB.
- Setup memory mapping (e.g. page table) and activate the execution of the new process.
- The PCB information contains registers (including stack pointer, program counter), page table, and other process-oriented OS information.
- The state change: Run->Ready for the old process and Ready->Run for the new process.

**Q1.2** Given the following c program that uses Linux `fork()/wait()` system calls

```
int main() {
    int pid, x=10, y;
    pid = fork();
    if (pid == 0) {
        y = 1;
        x = x + y;
        printf("x is %d. y is %d \n", x,y);
    } else {
        y = 2;
        x = x + y;
        wait(NULL);
        printf("x is %d. y is %d \n", x,y);
    }
}
```

```
x is 11, y is 1
x is 12, y is 2
```

**Q2.1** What is the main difference between a process and a thread?

Threads allocate separate stack space within the same process resource, share memory for code/data segments. Processes don't share memory normally. Separate resource (address space) is allocated from the system for each process. Each process is recognized by OS during resource allocation and scheduling while a thread created within a process may or may not be recognized by the OS during scheduling.

**Q2.2** Explain what Nachos OS does in the following four statements during Nachos thread forking.

```
Thread::Fork(VoidFunctionPtr func, int arg) {
    StackAllocate(func, arg);
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

- Allocate stack space and setup thread control block properly used for thread context switch (e.g. setup stack pointer, the thread starting address, and argument location).
- Disable interruption.
- Put this thread into a queue with a ready state Enable interruption.
- Enable interruption.

**Q3.1** What is a race condition?

Multiple threads or processes may operate on the same shared data at the same time; the result depends on their data access time and execution order.

**Q3.2** For Project 1, if we change `ThreadTest()` in file `threadtest.cc` as follows:

```
int x = 10;
void SimpleThread(int y) {
    currentThread->Yield();
    x = x + y;
    printf("x is %d. y is %d \n", x, y);
}
void ThreadTest() {
    Thread *t = new Thread("forked thread");
    t->Fork(SimpleThread, 1);
    SimpleThread(2);
}
```

In this case, Nachos threads only yield to each other when executing `Yield()`. List and explain the values of "x" and "y" printed in executing `ThreadTest()`.

```
x = 12, y = 2
x = 13, y = 1
```

The parent thread executes `SimpleTread(2)` first, but yields. The child thread starts `SimpleTread(1)`, but yields. Then parent thread continues and prints. After its completion, the child thread resumes and prints.

**Q4.1** What is the benefit of TLB for a paging scheme?

Speed up looking up of address translation using a fast hardware cache.

**Q4.2** Given a one-level paging scheme for memory address mapping, if TLB access takes 10 nanosecond and each physical memory access takes 100 nanoseconds, what will be the desired TLB hit ratio to have the effective memory access time within 120 nanoseconds to complete both logical address translation and physical memory data fetching?

Solve $10 + (1 - x) * 100 + 100 = 120$ for $x = 0.9$.

**Q5.1** Given a one-level paging scheme where physical memory is divided into a set of pages with uniform

size 8K bytes and each page table entry uses 4 bytes, what is the maximum size of a logical memory space for each process with this paging scheme? Explain your reason.

The page table is also 8K bytes, each entry uses 4 bytes, thus 8K/4 = 2K entries in a page table; thus there are total of 2K pages, the maximum size of memory space is 2K*8KB = 16MB.

**Q5.2** Design a 2-level paging scheme for a 38-bit logical space and illustrate the address translation steps. Each physical page is of size 16K bytes and each page table entry uses 4 bytes.

(12, 12, 14)

- Physical page size is 16K bytes, and offset for a page use log(16K) = 14 bits.
- Each physical page can host 16K/4 = 4K entries. Thus a page table using one physical page can map at most 4Kx16K=64MB physical space. A two-level table can map at most 64MB*4K=256GB space which is 2^38.
- The level-1 outer table has 4K entries, and log(4K)=12. That needs 12 bits to identify a particular entry. Thus the outer table part of the logical address needs 12 bits.
- Consequently the level-2 inner table part of logical address needs 12 bits. The scheme is 2-level paging and 38 bits will be split as 12, 12, and 14.

**Q6** The following pseudo code for a producer-consumer problem needs a synchronization.

```
// Consumer thread
while (1) { /* consume next data item */ }
// Producer thread
while (1) { /* produce next data item */ }
```

Extend the above code and use condition variables and a lock to synchronize the critical section and avoid busy waiting so that a producer waits if there is no buffer space to store data and a consumer waits if there is no data to consume.

```
// Consumer thread
while(1){
    lock->acquire();
    while (queue is empty) {
        condData->wait(lock)
    }
    Consume next data item;
    condSpace->signal(lock);
    lock->release();
}
// Producer thread
while (1) {
    lock->acquire();
    while (queue is full) {
        condSpace->wait(lock)
    }
    Produce next data item;
    condData->signal(lock);
    lock->release();
}
```

### F12 final

**Q1.1** What is the benefit of incorporating the copy-on-write feature in process forking?

Delay page duplication when creating a child process until this page is modified by the child process. If no pages are modified in the child process, and then cost of duplication can be eliminated.

**Q1.2** Given the following C program that uses Linux `fork()/wait()` system calls, Assume `printf()` is an atomic operation. Show all possible results printed when running a multiprogramming Linux system.

```
int main() {
    int pid, x = 100, y = 1;
    pid = fork();
    if (pid == 0) {
        x = x + 1;
        y = y + x;
        printf("x is %d. y is %d \n", x,y);
    } else {
        x = x + 2;
        y = y + x;
        printf("x is %d. y is %d \n", x,y);
    }
}
```

```
x is 101. y is 102
x is 102. y is 103

x is 102. y is 103
x is 101. y is 102
```

**Q2.1** Explain the advantages and disadvantages of Shortest-Job-First (SJF) scheduling. Explain why the Multi-Level-Feedback-Queue can be considered as an approximation to SJF while addressing the weakness of SJF?

- Advantage of SJF: reduce aveage job waiting time.
- Disadvantage of SJF: there could a starvation for some long jobs.
- In multi-queue scheduling, short jobs can be assigned in one queue with more resource. Long jobs can be assigned to another queue. Allocating resource proportionaly among queues avoids starvation.

**Q2.2** Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

The processes are assumed to have arrived in the order P1,P2,P3,P4,P5, all at time 0. Draw 3 charts illustrating the execution of these processes using FCFS (First-come-first-serve), SJF (shortest-job-first), and RR (round-robing with quantum=1) scheduling. What is the waiting time and turnaround time of process P1 for each of the scheduling algorithms?

- P1, arrival time = 0, burst time = 10
- P2, arrival time = 0, burst time = 1
- P3, arrival time = 0, burst time = 2
- P4, arrival time = 0, burst time = 1
- P5, arrival time = 0, burst time = 5
- FCFS: P1 P2 P3 P4 P5, turnaround = 10, waiting = 0
- SJF: P2 P4 P3 P5 P1, turnaround = 19, waiting = 9
- RR: P1 P2 P3 P4 P5 P1 P3 P5 P1 P5 P1 P5 P1 P5 P1, turnaround = 19, waiting = 9

**Q3.1** What is the role of dirty-bits in Nachos virtual memory implementation in Project 3? How does this help in improving execution performance.

Dirty bits indicate if a memory page has been modified. It can be used to check if a page should be written back the swapping store or not when a page is evicted from memory. It can save disk I/O and avoid unncessary writes.

**Q3.2** A demand-paging memory system translates an address using a TLB and a one-level page table in the main memory. The TLB hit ratio is 99% and it takes 0 nanosecond for TLB access. Each physical memory access takes 100 nanoseconds. If this virtual page is not in memory, then page fault handling takes 10,000 microseconds. What will be the maximum page fault rate that makes the overall average effective memory access time to be less than 1 microsecond?

If there is no page fault, average memory access time in ms is $0.99 * (0 + 0.1) + 0.01 * (0.1 + 0.1) = 0.101$ . Then solve $(1 - f) * 0.101 + f(0.101 + 10000) = 1$ for $f = 0.008999$.

**Q3.3** Given 3 empty physical pages, draw figures to illustrate how a page replacement scheme deals with the following reference string (1, 2, 3, 4, 2, 5, 2) using the LRU strategy. Compute the number of page faults.

Five faults: 1F, 2F, 3F, 4F(replace 1), 2, 5F(replace 3), 2.

**Q3.4** Given the following C program which access a 2D integer array `a[1024][1024]`. A demand-paging memory is divided into a set of uniform 1K byte pages. Each integer uses 4 bytes. Assume that the binary instructions for the following program are always in separate memory space. The LRU page replacement algorithm is used to manage a 512KB memory that stores the array. Initially data array `a[][]` is not in memory. Derive and explain the number of page faults in running this program. If the memory increases to 1MB, what is the number of page faults in running this program?

```
for (i = 0; i < 1024; i++) for (j = 0; j < 1024; j++) a[j][i] = 100;
```

The C program stores this array in a row-wise format. Each row needs 1024*4/1K = 4 memory pages. 512KB memory means 512 pages. The code accesses column 1, requiring the loading of 1K pages with 1K page faults. For column 2, old data loaded is swapped out, and thus it takes another 1K page faults.

In summary, 1024x1024 page faults occur.

If memory is 1MB, then memory can hold 1K pages. After column 1 access, data in memory can be used for column 2 access. Thus total faults will be 1024*4 = 4K.

**Q4.1** What are advantages and disadvantages of RAID 0 and RAID 1?

RAID 0 improves access bandwidth (paralell read or write), but there is no extra redudancy added to deal with a disk failure. RAID 1 improves reliability and can torelate one disk fail, and it doubles read throughput. But 50% of disk space is wasted.

**Q4.2** The original Nachos file system implementation (before Project 3 extension) uses a single-level indexed allocation. Given a file currently consisting of 1 disk block, how many disk block read or write operations are required to add a block to the end of this file?

Total 2 operations. 1 IO operation to load the file header. Then the kernel modifies the file header to add one block. Then it does 1 IO operation to store the added data block. (If they answer 3, then the third has to be writing the index header to the disk. Then it is fine).

**Q4.3** We use indexed allocation to store a file on a disk and we assume that each disk block is of size 16K bytes. Each entry in an index block takes 4 bytes. What is the maximum size of a file that can be mapped by a single-level indexed allocation? Illustrate with a figure to show the levels of index indirection required to map a file of size 16 gigabytes?

- 16K/4=4K entries per index block.
- The maximum size of a file with single indirect block = 4K x 16K bytes = 64MB.
- Maximum size for two levels: 4K x 4K x 16K = 256GB.
- 16GB would require 2 levels.

**Q6.1** Suppose that a disk drive has 200 cylinders, numbered from 0 to 199. The drive is currently serving a request at cylinder 120, and the previous request was at cylinder 100. The queue of pending requests, in FIFO order is: 80, 190, 40, 110. Illustrate the disk arm movement using the FCFS and SCAN algorithms. Compute the total movement in cylinders for each algorithm.

- FCFS: 120, 80, 190, 40, 110. In total 370.
- SCAN: 120, 190, 110, 80, 40. In total 220.

**Q6.2** Suppose this disk drive spins at 12000 RPM, and has a sector size of 512 bytes, and holds 50 sectors per track. What is sustained transfer rate of this drive in megabytes per second? Assume average seek time for the above drive is 7.3 milliseconds, what is the effective transfer rate for random-access of two consecutive sectors on a disk track? (Namely find the position first, then transfer two sectors together)

- Each second spins 12000/60 = 200 times.
- Each spin transfers a track of 25 KB (50 sectors x 0.5 KB).
- Sustained average transfer rate is 200 x 25 KB = 5 MB/s.
- Average rotational cost is time to travel track: 1/100 x 0.5 = 2.5 ms.
- Transfer time is 7.3 ms to seek + 2.5 ms rotational latency + 0.2 ms (reading two sectors takes 0.001 MB / 5 MB) = 10 ms.
- Effective transferring rate: 1KB / 0.01s = 0.1 MB/s.

**Q7** The following pseudo code is executed by each thread and the code uses `Swap()` atomic instruction to implement a critical section solution to synchronize threads.

```
ThreadFunction() {
    int key = TRUE;
    int lock = TRUE;
    while (key == TRUE)
        Swap (&lock, &key);
    // Perform critical section code
    lock = FALSE;
}
```

**Q7.1** Should variables `lock` and `key` be shared among threads? Correct the above code and initialize variables properly.

lock is shared, key is not.

```
int lock = FALSE;
ThreadFunction() {
    int key = TRUE;
    while (key == TRUE)
        Swap (&lock, &key);
    // Perform critical section code
    lock = FALSE;
}
```

**Q7.2** Does the above corrected solution satisfy each of the following three properties: mutual exclusion, progress, bounded waiting?

Mutual exclusion: yes. Progress: yes. Bounded waiting: no.

**Q7.3** Chapter 5 of the text book discussed a solution for the readers-writers problem. In this problem, a data set is shared among a number of threads. A synchronization solution is required so that multiple readers can read at the same time. But when one writer is accessing a shared data item, other writers or readers cannot access this shared item.

```
Lock *w = new Lock("Write-Lock");

// Writer thread:
while(1) {
  w->Acquire();
  // Writing is performed
  w->Release();
}
```

```
// Reader thread:
while(1) {
  w->Acquire();
  // Reading is performed
  w->Release();
}
```

Explain if the following solution meets the requirement? If not, explain and provide a solution with pseudo code.

```
Semaphore mutex initialized to 1 (they may use a lock)
Semaphore wrt initialized to 1 (it is wrong to use a lock for wrt)
Integer readcount initialized to 0

// Writer thread
while(1) {
  wrt->P();
  // Writing is performed
  wrt->V();
}

// Reader thread
while(1) {
  mutex.P();
  readcount++;
  if (readcount == 1) wrt.P();
  mutex.V()
  // reading is performed
  mutex.P();
  readcount--;
  if (readcount == 0) wrt.V();
  mutex.V();
}
```

Extend this solution and allow at most 10 readers that can read at the same time.

```
Semaphore mutex initialized to 1
Semaphore wrt initialized to 1
Semaphore rd initialized to 10
Integer readcount initialized to 0

// Writer thread
while(1) {
    wrt->P();
    // Writing is performed
    wrt->V();
}

// Reader thread
while(1) {
    rd.P();
    mutex.P();
    readcount++;
    if (readcount == 1) wrt.P();
    mutex.V()
    // reading is performed
    mutex.P();
    readcount--;
    if (readcount == 0) wrt.V();
    mutex.V();
    rd.V();
}
```

```
Lock mutex
Semaphore wrt initialized to 1
Integer readcount initialized to 0
Cond as condition variable

// Writer thread
while(1) {
    wrt->P();
    // Writing is performed
    wrt->V();
}

// Reader thread
while(1) {
    mutex.Acquire();
    while (readcount >= 10) Cond->Wait(mutex);
    readcount++;
    if (readcount == 1) wrt.P();
    mutex.Relase();
    // reading is performed
    mutex.Acquire();
    readcount--;
    Cond->Signal(mutex);
    if (readcount == 0) wrt.V();
    mutex.Relase();
}
```

## Code

**RWLock implementation**

```
//----------------------------------------------------------------------
// Semaphore::Semaphore
//  Initialize a semaphore, so that it can be used for synchronization.
//
//  "debugName" is an arbitrary name, useful for debugging.
//  "initialValue" is the initial value of the semaphore.
//----------------------------------------------------------------------

Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}

//----------------------------------------------------------------------
// Semaphore::~Semaphore
//  De-allocate semaphore, when no longer needed.  Assume no one
//  is still waiting on the semaphore!
//----------------------------------------------------------------------

Semaphore::~Semaphore()
{
    delete queue;
}

//----------------------------------------------------------------------
// Semaphore::P
//  Wait until semaphore value > 0, then decrement.  Checking the
//  value and decrementing must be done atomically, so we
//  need to disable interrupts before checking the value.
//
//  Note that Thread::Sleep assumes that interrupts are disabled
//  when it is called.
//----------------------------------------------------------------------

void
Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interru

    while (value == 0) {            // semaphore not available
        queue->Append((void *)currentThread);   // so go to sleep
        currentThread->Sleep();
    }
    value--;                    // semaphore available, consume its value
```

```
    (void) interrupt->SetLevel(oldLevel);   // re-enable interrupts
}

//----------------------------------------------------------------------
// Semaphore::V
//  Increment semaphore value, waking up a waiter if necessary.
//  As with P(), this operation must be atomic, so we need to disable
//  interrupts.  Scheduler::ReadyToRun() assumes that threads
//  are disabled when it is called.
//----------------------------------------------------------------------

void
Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)    // make thread ready, consuming the V immediat
        scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Lock::Lock
//  Initialize a lock, so that it can be used for synchronization.
//  A lock is very much like a semaphore with initial value 1.
//p
//  "debugName" is an arbitrary name, useful for debugging.
//----------------------------------------------------------------------

Lock::Lock(char* debugName)
{
    name = debugName;
    value = 1;
    queue = new List;
}

//----------------------------------------------------------------------
// Lock::~Lock
//   Dedestuct a lock, when no longer needed. Assume no one is still
//   holding a lock!
//----------------------------------------------------------------------
Lock::~Lock()
{
    delete queue;
}

//----------------------------------------------------------------------
// Lock::Acquire
//   This function waits for a lock to become free and then acquires
//   the lock for the current thread.
//----------------------------------------------------------------------
void Lock::Acquire()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // if the lock is acquired by someone else, put the current thread
    // to the waiting queue and sleep
    while (value == 0) {
        queue->Append((void*)currentThread);
        currentThread->Sleep();
    }
    // acquire the lock
    value = 0;
    owner = currentThread;
    (void)interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Lock::Release
//   This function releases a lock that was previously acquired by the
//   current thread, and wakes up one of the threads waiting for the
//   lock.
//----------------------------------------------------------------------
void Lock::Release()
{
    ASSERT(isHeldByCurrentThread());
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // release the lock
    value = 1;
    owner = NULL;
    // put a thread in the waiting queue (if any) to the ready queue
    Thread *thread = (Thread*)queue->Remove();
    if (thread != NULL)
        scheduler->ReadyToRun(thread);
    (void)interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Lock::isHeldByCurrentThread
//   True if the current thread holds this lock. It is useful for
//   checking in Release, and in condition variable ops below.
//----------------------------------------------------------------------
bool Lock::isHeldByCurrentThread()
{
    return value == 0 && currentThread == owner;
}


//----------------------------------------------------------------------
// Condition::Condition
//   Initializes a condition object.
//----------------------------------------------------------------------
Condition::Condition(char* debugName)
{
    name = debugName;
    queue = new List;
}

//----------------------------------------------------------------------
// Condition::~Condition
//   Destruct a condition object, when it is no longer needed.
//----------------------------------------------------------------------
Condition::~Condition()
{
    delete queue;
}

//----------------------------------------------------------------------
// Condition::Wait
//   Waits for a condition to become free and then acquires the
//   condition for the current thread.
//----------------------------------------------------------------------
void Condition::Wait(Lock* conditionLock)
{
    ASSERT(conditionLock->isHeldByCurrentThread());

    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    conditionLock->Release();
    queue->Append((void*)currentThread);
    currentThread->Sleep();
    conditionLock->Acquire();
    (void)interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Condition::Signal
//   Wakes up one of the threads that is waiting on the condition.
//----------------------------------------------------------------------
void Condition::Signal(Lock* conditionLock)
{
    ASSERT(conditionLock->isHeldByCurrentThread());

    Thread *thread = (Thread*)queue->Remove();
    if (thread != NULL)
        scheduler->ReadyToRun(thread);
}

//----------------------------------------------------------------------
// Condition::Broadcast
//   Wakes up all threads that are waiting for the condition.
//----------------------------------------------------------------------
void Condition::Broadcast(Lock* conditionLock)
{
    ASSERT(conditionLock->isHeldByCurrentThread());

    Thread *thread;
    while ((thread = (Thread*)queue->Remove()))
        scheduler->ReadyToRun(thread);
}
```

**RWLock usage**

```
#include "rwlock.h"
#pragma GCC diagnostic ignored "-Wwrite-strings"

RWLock::RWLock() {
#ifdef P1_SEMAPHORE
    read_lock = new Semaphore("read lock", 1);
    write_lock = new Semaphore("write lock", 1);
    blocking_writer = 0;
#endif

#ifdef P1_LOCK
    read_lock = new Lock("read lock");
    write_lock = new Lock("write lock");
    blocking_writer = 0;
#endif

#ifdef P1_RWLOCK
    active_writer = 0;
    active_reader = 0;
    waiting_writer = 0;
    waiting_reader = 0;
    mutex = new Lock("mutex");
    ok_to_read = new Condition("read cv");
    ok_to_write = new Condition("write cv");
#endif
}
```

```
RWLock::~RWLock() {
#ifdef P1_SEMAPHORE
    delete read_lock;
    delete write_lock;
#endif

#ifdef P1_LOCK
    delete read_lock;
    delete write_lock;
#endif

#ifdef P1_RWLOCK
    delete mutex;
    delete ok_to_read;
    delete ok_to_write;
#endif
}

void RWLock::startWrite() {
#ifdef P1_SEMAPHORE
    write_lock->P();
#endif

#ifdef P1_LOCK
    write_lock->Acquire();
#endif

#ifdef P1_RWLOCK
    mutex->Acquire();
    while (active_writer + active_reader > 0) {
        waiting_writer++;
        ok_to_write->Wait(mutex);
        waiting_writer--;
    }
    active_writer++;
    mutex->Release();
#endif
}

void RWLock::doneWrite() {
#ifdef P1_SEMAPHORE
    write_lock->V();
#endif

#ifdef P1_LOCK
    write_lock->Release();
#endif

#ifdef P1_RWLOCK
    mutex->Acquire();
    active_writer--;
    if (waiting_writer > 0)
        ok_to_write->Signal(mutex);
    else
        ok_to_read->Broadcast(mutex);
    mutex->Release();
#endif
}

void RWLock::startRead() {
#ifdef P1_SEMAPHORE
    read_lock->P();
    blocking_writer++;
    // first reader locks write_lock
    if (blocking_writer == 1)
        write_lock->P();
#endif

#ifdef P1_LOCK
    read_lock->Acquire();
    blocking_writer++;
    // first reader locks write_lock
    if (blocking_writer == 1)
      write_lock->Acquire();
#endif

#ifdef P1_RWLOCK
    mutex->Acquire();
    while (active_writer > 0 || waiting_writer > 0) {
        waiting_reader++;
        ok_to_read->Wait(mutex);
        waiting_reader--;
    }
    active_reader++;
    mutex->Release();
#endif
}

void RWLock::doneRead() {
#ifdef P1_SEMAPHORE
    blocking_writer--;
    // last reader unlocks write_lock
    if (blocking_writer == 0)
        write_lock->V();
    read_lock->V();
#endif

#ifdef P1_LOCK
    blocking_writer--;
    // last reader unlocks write_lock
    if (blocking_writer == 0)
      write_lock->Release();
    read_lock->Release();
#endif

#ifdef P1_RWLOCK
    mutex->Acquire();
    active_reader--;
    if (active_reader == 0 && waiting_writer > 0)
        ok_to_write->Signal(mutex);
    mutex->Release();
#endif
}
```

# Project README

### Project 2 Multiprogramming

#### First part

In the first part of this assignment, you are to implement the `Fork()`, `Yield()`, `Exit()`, `Exec()`, and `Join()` system calls that act as follows:

- The `Fork(func)` system call creates a new user-level (child) process, whose address space starts out as an exact copy of that of the caller (the parent), but immediately the child abandons the program of the parent and starts executing the function supplied by the single argument. Notice this definition is different from the one in the `syscall.h` file in Nachos.
- The `Yield()` call is used by a process executing in user mode to temporarily relinquish the CPU to another process.
- The `Exit(int)` call takes a single argument, which is an integer status value as in Unix. The currently executing process is terminated and returns the exit code.
- The `Exec(filename)` system call spawns a new user-level process with a new address space that begins executing a new program given by the object code in the Nachos file whose name is supplied as an argument to the call. It should return to the parent a `SpaceId` which can be used to uniquely identify the newly created process. `syscall.h` defines 4 arguments and please ignore the other 3 arguments.
- The `Join()` call waits and returns only after a process with the specified ID (supplied as an argument to that call) has finished. It collects and returns the exit code from the targeted process.

#### Second part

In the second part of this assignment, you are to implement the file system calls: `Creat()`, `Open()`, `Read()`, `Write()`, and `Close()`. The semantics of these calls are specified in `syscall.h`. You should extend your file system implementations to handle the console as well as normal files.

#### Issues to consider

Here is an outline of the the major issues you will have to deal with to make Nachos into a multiprogrammed system:

In order to handle the various system calls, you will need to modify the `ExceptionHandler()` function in `exception.cc` to determine which system call or exception occurred, and to transfer control to an appropriate function. You might want to consider introducing "stubs" (functions with empty bodies or with debugging printout so you can tell when they are called) for all the system calls right away, and then postpone their actual implementation until a bit later. This strategy will help you understand better how control is transferred from user mode to system mode when a system call is executed.

The Nachos code you have been given is extremely simple-minded about memory management. In particular, the constructor function `AddrSpace::AddrSpace()` simply determines the amount of memory that will be required by the application to be run and then allocates that much space contiguously starting at address zero in physical memory. The page tables (which control the address translation hardware) are set up so that the logical addresses (what the user program sees) are identical to the physical addresses (where the data is actually stored).

The above scheme is inadequate for running more than one application at a time. You will need to design and implement a scheme for allocating and freeing physical memory, and you will need to arrange to set up the page tables so that the logical address space seen by a user application is a contiguous region starting from address zero, even though the data is stored at different physical addresses. You will want to implement a memory management scheme that is flexible enough to extend to virtual memory later in the semester. We suggest implementing a C++ class with methods for allocating and freeing physical memory one page at a time. By setting up the page tables properly, you can give the user application a contiguous logical address space even though each page of actual data might be stored anywhere in physical memory.

The `Fork()` system call is the most difficult part of this assignment. It is different from the system call `Exec` in that `Fork` will start a new process that runs a user function specified by the argument of the call, while `Exec` will start a process that runs a different executable file. The parameter types for `Fork()` and `Exec()` also differ. `Fork(func)` takes an argument func which is a pointer to a function. The function must be compiled as part of the user program that is currently running. By making this system call `Fork(func)`, the user program expects the following: a new thread will be generated for use by the user program; and this thread will run func in an address space that is an exact copy of the current one. This implementation of Fork makes it possible to have and to access multiple entry points in an executable file.

To make the system call `Fork(func)` work for the user program, you will need to know how to find the entry point of the function that is passed as the parameter. The parameter convention is determined by the cross-compiler which produces executable code from the user source program. Look at the file `exception.cc` to see that this entry point, which is an address in the executable code's address space, is already loaded into register 4 when the trap to the exception handler occurs. All you need to do is to insert code into the exception handler (or call a new function of your own) which does the following: set up an address space which is a copy of the address space of the current thread, and load the address that is in register 4 into the program counter. After these steps, use `Thread::Fork()` to create a new thread,

initialize the MIPS registers for the new process, and have both the new and old processes return to user mode. The parent should return to user mode by returning from the exception handler, the child process should continue to run from the address that is now in the program counter, which is the entry point of the function. To implement Fork, you will need to introduce modifications to the AddrSpace class in addrspace.cc so that you can make a
`clone'' of a running user application program. We suggest adding a functi
function.

The `Exit()` system call should work by calling `Thread::Finish()`, but only after deallocating any physical memory and other resources that are assigned to the thread that is exiting.

In order to implement the `Exec()` system call, you will need a method for transferring data (the name of the executable, supplied as the argument to the system call) between the user address space and the kernel. You are not to use functions `Machine::ReadMem()` and `Machine::WriteMem()` in `machine/translate.cc`. Instead, you will have to code your own functions that take into account the address translations described by the page tables to locate the proper physical address for any given logical address. (Recall that strings in C are stored as sequences of characters in successive memory locations, terminated by a null character.)

Once the name of the executable has been copied into the kernel, and the file has been verified to exist, the executable file should be consulted to determine the amount of physical memory required for the new program. This physical memory should be allocated and initialized with data from the executable file, the page tables thread should be adjusted for the new program, the MIPS registers should be reinitialized for starting at the beginning of the new program, and control should return to user mode. File progtest.cc contains a sample for executing a binary program.

If you use `machine->Run` to execute a user program, it terminates the current thread. Since `Exec()` needs to return a space ID to the caller, you should find a way to do that.

NOTE: The object code produced by the MIPS cross-compiler assumes that the data segment begins at the physical address immediately following the text segment. In particular, there is no page alignment, so that if the text segment ends in the middle of a page, then the data segment will start just after it and the page will contain both code and data.

`Yield()` system call will call `Thread::Yield()` after making sure to save any necessary state information about the currently executing process.

Be sure to synchronize your code correctly. You will need to put lock operations in your code to ensure that it will work properly. Your locking should be fine grained enough to eliminate any spurious latency problems caused by coarse grained locking. For example, any time a thread accesses the disk or the console it should not hold a lock that would prevent another thread from accessing some other I/O device or piece of data.

You should buffer user file reads in a disk buffer called diskBuffer (defined in `system.cc`). All of your user-level file I/O must go through the `diskBuffer`.

You will need to solve a synchronization problem that occurs when multiple processes try to read or write from a file at the same time.

**Threads and processes**

`Thread::Fork()` spawns a new kernel thread that uses the same AddrSpace as the thread that spawned it. If that address space is duplicated and not shared, it is no longer a kernel thread but a Forked Process. If that AddrSpace instead contains code and data loaded in from a separate file, it is no longer a forked process, but an executed process. The details of `Fork()` and `Exec()` are covered in steps 1) and 7) below.

**System calls**

In `code/userprog/exception.cc`, put function calls for each system call. Just print debug statements in these functions for now, so you can see when system calls get executed. The function may need to return or take an argument. Because the argument lies in user space, you will need to transfer it over using machine register reads and writes. A sample stub illustrating this is shown below.

```
case SC_Join:
    int result = myJoin(machine->ReadRegister(4));
    machine->WriteRegister(2, result);
    break;
```

If the system calls: `Fork()`, `Yeild()`, `Exec()`, `Join()` and `Exit()` are implemented in that order, you will not have to worry about the call you are currently working on depending upon unimplemented calls.

Remember that the last thing the `ExceptionHandler` function needs to do when executing a system call is increment the program counter. Write a helper function to do this - it needs to update `PCreg`, `NextPCreg`, and `PrevPCreg`. They should all incriment by 32 bits (4 bytes).

**10 steps to a multi-programmed Nachos**

1. Implement `Fork()`. Fork will create a new kernel thread and set it's AddrSpace to be a duplicate of the CurrentThread's space. It sets then `Yields()`. The new thread runs a dummy function that will will copy back the machine registers, PC and return registers saved from before the yield was performed. You did save the PC, return and other machine registers didn't you?

   Duplicating the AddrSpace requires the implementation of a Memory Manager detailed in steps 2-4. `Fork()` will not work completely until the completion of step 4. Don't get stuck on step 1), steps 2-4 are much more important.

2. Write a Memory Manager that will be used to facilitate contiguous virtual memory. The amount of memory available to the user space is the same as the amount of physical memory, it's not until project 3 that you will have to implement swapping virtual memory.

   You will need just two methods at first 1) `allocFrame()` allocates the first free frame (in physical memory) and 2) `freeFrame(int i)` takes the index of a frame and marks it free. You can use a `Bitmap` (in `code/userprog/bitmap.*`) with one bit per page to track allocation or use your own integer array, which ever you prefer.

   Modify `AddrSpace` (`code/userprog/addrspace.*`) to use the memory manager. first, modify the page table constructors to use pages allocated by your memory manager for the physical page index. The later modification will come in step 4.

3. Write the`AddrSpace::Translate` function, which converts a virtual address to a physical address. It does so by breaking the virtual address into a page table index and an offset. It then looks up the physical page in the page table entry given by the page table index and obtains the final physical address by combining the physical page address with the page offset. It might help to pass a pointer to the space you would like the physical address to be stored in as a paramter. This will allow the function to return a boolean TRUE or FALSE depending on whether or not the virtual address was valid. If confused, consult the text book on memory management and page table or `Machine::Translate()` in `machine/translate.cc`.

4. Write the `AddrSpace::ReadFile` function, which loads the code and data segments into the translated memory, instead of at position 0 like the code in the AddrSpace constructor already does. This is needed not only for `Exec()` but for the initial startup of the machine when executing any test program with virtual memory.

   You should buffer user file reads in a disk buffer called diskBuffer (defined in `system.h`). All of your user-level file I/O must go through the diskBuffer. Be sure to to under or over run the buffer during the copy. Also be sure not to write too much of the file into memeory. You can use the following prototype for the function.

   ```
   int AddrSpace::ReadFile(int virtAddr,
                           OpenFile* file,
                           int size,
                           int fileAddr);
   ```

   You will also need to use the functions: `File::ReadAt(buff,size,addr)` and `bcopy(src,dst,num)` as well as the memory locations at `machine->mainMemory[physAddr]`.

   At this point, test programs should work the same as before. That is, the halt program and other system calls will still operate the way they did before you modified AddrSpace. Also `Fork()` should be working. Test your implementation appropriately.

5. Write the PCB and a process manager. Create a PCB class that will store the necessary information about a process. Don't worry about putting everything in it right now, you can always add more as you go along. To start, it should have a `PID`, `parent PID`, and `Thread*`. The process manager should do the same thing as the memory manager - it has `getPID` and `clearPID` methods, which return an unused process id and clear a process id respectively. Again, use a bitmap or similar integer array. You'll also need an array of `PCB*` to store the PCBs. Modify the AddrSpace constructors to include a PCB as an attribute.

6. With the process manager implemented you will have to modify `progtest.cc` so that the first process started by the system is given a PCB. The function `StartProcess()` is defined in `progtest.cc` and called from `threads/main.cc`. It loads and runs the first program (the one you passed in with -x). Modify `StartProcess()` so that a PCB is created for the first process. This process won't have a parent -- be sure to handle this accordingly.

7. Implement `Yield()`. Given that forked processes are almost the same as kernel threads, this one should be trivial.

8. Implement `Exec()`. Exec is creating a new process (kernel thread) with new code and data segments loaded from the OpenFile object constructed from the filename passed in by the user. In order to get that file name you will have to write a function that copies over the string from user space. This function will start copying memory from the physical address pointed to by the virtual address in `machine->ReadRegister(4)`. It should go until it hits a `NULL` byte.

   Fork the new thread to run a dummy function that sets the machine registers straight and runs the machine. The calling thread should yield to give control to the newly spawned thread. At this point you should be able to call test files from other test using `Exec(someTestFile)` from `someOtherTestFile.c` (in the `test/` dir).

9. Implement `Join()`. Join should force the current running thread to wait for some process to finish. The PCB manager can keep track of who is waiting for who using a condition variable for each PCB.

10. Implement `Exit(int status)`. This function should set set the status in the PCB being exited. It should also force any threads waiting on the exiting process to wake up.

11. Test your system calls using your own test programs, then the ones in `~cs170/nachos-projtest/proj2` that don't rely on part 2. If you have time, test using some programs to do crazy or malicious things.

**Part 2 implementation suggestions**

We will be using Nachos files exclusivly in part 2. This will be good preparation for project 3. Nachos files are similar to Unix files except they are stored on avirtual disk that is implemented as one big Unix file. The interface to `Create`, `Open`, `Close`, `WriteAt` and `ReadAt` to those files are defined in `filesys/filesys.cc`, `filesys/openfile.cc` and`filesys/filehdr.h`. You may want to look through those files when getting ready to call these functions for the first time.

## 10 steps to an I/O enabled Nachos

1. Create a `SysOpenFile` object class that contains a pointer to the file system's `OpenFile` object as well as the systems `FileID` and `fileName` for that file and the number of user processes accessing currently it. Declare an array of `SysOpenFile` objects for use by all system calls implemented in part 2.

2. Create a `UserOpenFile` object class that contains the `fileName`, an index into the global `SysOpenFile` table and an integer offset represeting a processes current position in that file.

3. Modify the `AddrSpace`'s' PCB to contain an array of `OpenUserFiles`. Limit the number to something reasonable, but greater than 20. Write a method (in `PCBManager`) that returns an `OpenUserFile` object given the filename.

4. Implement `Create(char *fileName)`. This is a straight forward call that should simply get the `fileName` from user space then use `fileSystem->Create(fileName,0)` to create a new instance of an `OpenFile` object. Until a user opens the file for IO it is not necessary to do anything further.

5. Implement `Open(char *fileName)`. This function will use an `OpenFile` object created previously by `fileSystem->Open(fileName)`. Once you have this object, check to see if it is already open by some other process in the global `SysOpenFile` table. If so, incriment the `userOpens` count. If not, create a new `SysOpenFile` and store it's pointer in the global table at the next open slot. You can obtain the `FileID` by looking up the name in your `SysOpenFile` table.

   Then create a new instance of an `OpenUserFile` object (given a `SysOpenFile` object) and store it in the current thread's PCB's `OpenUserFile` array.

   Finally, return the `FileID` to the user.

6. Implement a function to read/write into `MainMem` and a buffer given a staring virtual address and a size. It should operate in the same way `AddrSpace::ReadFile` writes into the main memory one `diskBuffer` at a time. It may help to put the section of the code in `ReadFile` into a *helper* function called `userReadWrite()` that is general enough that both `ReadFile()`, `myRead()` and `myWrite()` can call. It need only be parameterized by the type of operation to be performed (`Read` or `Write`).

   When called by `Write()`, it will read from the `MainMem` addressed by the virtual addresses. It writes into the given (empty) buffer. `Write()`, will then put that buffer into an `OpenFile`. When called by `Read()`, It will write into `MainMem` the data in the given (full) buffer that `Read()` read from an `OpenFile`.

7. Implement `Write(char *buffer, int size, OpenFileId (int)id)`. First you will need to get the arguments from the user by reading registers 4-6. If the `OpenFileID` is `ConsoleOutput` (defined in `syscall.h`), then you call `PutChar()` in `console.cc` to print the buffer content. Otherwise, grab a handle to the `OpenFile` object from the user's openfile list pointing to the global file list. Why can't you just go directly to the global file list? Becuase the user may not have opened that file before trying to write to it. Once you have the `OpenFile` object, you should fill up a buffer using your `userReadWrite()` function. Then simply call `OpenFileObject->Write(myOwnBuffer, size);`

8. Implement `Read(char *buffer, int size, OpenFileId id)`. Get the arguments from the user in the same way you did for `Write()`. If the `OpenFileID` is `ConsoleInput`, use a routine (`GetChar()` in `console.cc`) to read into a buffer one character at a time. Otherwise, grab a handle to the `OpenFile` object in the same way you did for `Write()` and use `OpenFileObject->ReadAt(myOwnBuffer,size,pos)` to put *n* characters into your buffer. `pos` is the position listed in the `UserOpenFile` object that represents the place in the current file you are writing to. The number read is returned from `ReadAt()`.

   Now that your buffer is full of the read, you must write that buffer into the user's memory using the `userReadWrite()` function. Finally, return the number of bytes written.

9. Test your system calls using your own test programs, then the ones in `~cs170/nachoos-projtest/proj2` Now that both parts are finished all test programs should work.

10. Test some more.

**FAQ**

> **Question**: How can I test part one (why doesn't printf work in my test programs)?

**Answer**: Until you have part 2 done you will have to test part 1 based on flow of execution. For example: to test if Fork/Exec works you Fork/Exec a function/executable and then call Halt() in the function/executable. If Nachos halts when you run your test program then your system call is probably working. Once you have the Write system call implemented you can further test part 1 by putting your Write statements where ever you wish to print something out. This will be useful for testing Join.

> **Question**: What does a process control block contain?

**Answer**: A process control block (pcb) contains the attributes of a process. Some of the major attributes are the thread, the pid (SpaceID), open files, etc (remember that the thread has a pointer to the addrspace which is an indirect attribute of a pcb). There should be a global table of process control blocks as well. Remember that you will also want to be able to get a particular process's condition so that it can be waited on in Join (if necessary) and broadcast in Exit. It may not seem like you are using the process control blocks much in part 1 of the project (except for adding and deleting them) but you will use them more in part 2 of the project).

> **Question**: How do I translate the name of the executable when implementing Exec?

**Answer**: You will want to put a translate function in addrspace that is similar to the translate function in Machine. You will use this function to translate a virtual address into a physical address (one page at a time).

> **Question**: Is there a difference between the parameters of Exec and Fork?

**Answer**: Yes there is. Exec takes the string representing the relative path (from where you run Nachos - userprog in this case) to the test program you wish to exec [ie. Exec("../test/myExecedProgram")]. Fork takes the name of the function you wish to fork. It is not a string, but a function pointer so it has the form Fork(myFunction) where you have implemented void myFunction() previously in you test program. In Exec you are translating the string that is passed in and in Fork you are using the pointer to myFunction as the PCReg value for when you run the new thread. If you try to pass a string to Fork or a name (not in string form) to Exec you will have serious problems

> **Question**: Does a Forked thread use the same addrspace as the thread who Forked it?

**Answer**: No it does not. It uses a COPY of the addrspace from the thread who Forked it. This means you must create a new addrspace that has the same size page table as the Forking thread and then you must copy the Forking thread's pages in memory into the Forked threads pages in memory.

> **Question**: The project spec says we need to add a function ReadFile to Addrspace. What is this used for?

**Answer**: This is used for copying the executable's code and data segments into memory (in the constructor for Addrspace (that Exec calls). noffH.code.virtualAddr is the logical address of the executable's code segment and noffH.initData.virtualAddr is the logical address of the executable's data segment. You no longer want to zero out (bzero) the memory. You need to copy these sections page by page into main memory. You will use your Addrspace::Translate to translate the logical data/code address to the physical address (in memory). Then you can use the C/C++/Unix "bcopy" to copy it over.

> **Question**: Where should we put the global structures/classes?

**Answer**: You can put these in system.h in the threads directory. Only put the global variables (their declaration) in this file. Make sure you actually instantiate the global structures (like memory manager) in the system.cc file in the initialize function under the correct #ifdef's (for userprog in this case).

> **Question**: Does the Exit system call take a parameter? What is that parameter?

**Answer**: The Exit system call should take an integer parameter. This parameter is the exit value of the process. This is the same as the exit values in Unix (0 -> good, 1 -> bad). For our purposes you can use any integer here. It only matters that if another process was "Join"ing on your process that the Join call would return the value that your process exited with (as per the project spec). This means that you need to save your exit value some how in the Exit system call so that Join can return it if necessary (even if you've already exited when someone calls Join on your process).

> **Question**: What should we do when we can't allocate enough physical pages in Addrspace constructor for a new thread?

**Answer**: You should let Fork/Exec know that you don't have enough memory so that it can let the user know (return pid of -1) that it didn't Fork/Exec a new process. Make sure that you check if there is enough free pages in physical memory to facilitate the number of pages for your new process before trying to allocate the physical pages for each logical page. If you do not do this one of your allocates will return -1 and you will have to deallocate all the pages you just allocated before you can let Fork/Exec know of the error.

> **Question**: Do Exec and Fork call thread->Fork?

**Answer**: Yes they do. After you have set all other information up (as in the project spec) you need to thread->fork a dummy function (that's in exception.cc). In this dummy function you will want to initialize and restore the registers (for Fork and Exec) and then set up the PC registers and return register address (for Exec only). After this you will call machine->Run() from the dummy function (for both Fork and Exec).

## Project 3 Virtual memory management

#### Additional information

In this assignment, we will leverage the following three bits maintained in each page table entry defined in `machine/translate.h`: **valid**, **dirty**, and **use** bits.

The valid bit indicates if a page is resident in physical memory or not. If the valid bit in a particular page table entry is set, the MIPS machine assumes that the corresponding virtual page is loaded into physical memory in the physical page frame specified by the physicalPage field. Whenever the program generates an access to that virtual page, the MIPS machine accesses the physical page. If the valid bit is not set, the system generates a page fault exception whenever the program accesses that virtual page. Your exception handler will then find a free physical page frame, read the page in from the backing store SWAP to that physical page frame, update the page table to reflect the new virtual to physical mapping, and restart the program. The MIPS machine will then resume the execution of the program at the instruction that generated the fault. This time the access should go through, since the page has been loaded into physical memory.

The dirty bit indicates if this page has been modified or not. To find a free frame, your page fault handler may need to eject an in-memory victim page. If the ejected victim page has been modified in the physical memory indicated by the dirty bit in the corresponding page table entry, the page fault handling code must write the page out to the backing store before reading the accessed page into its physical page frame.

The use bit indicates if this page has been recently accessed. On a page fault, the kernel must decide which page to replace; ideally, it will throw out a page that will not be referenced for a long time, keeping pages in memory those that are soon to be referenced. You are asked to implement the second chance algorithm leveraging this use bit. This bit is set to `TRUE` every time a user program reads or writes this page. The second chance algorithm follows a clockwise order to find the next victim whose use bit is `FALSE`. If it sees a page with the use bit value `TRUE`, it gives this page the second chance to stay, and the use bit of this page should be set as `FALSE`.

**Q&A**

> Where to modify for Task 1?

For this part, you will extend `virtualmemorymanager.cc` to rewrite Function `swapPageIn()`.

> What are relevant files to read?

Files in `vm` directory, `userprog` and `machine` directories. The given starter code is one possible approach and other designs are also possibles.

> Where to start?

First, understand how address translation works in method `Machine::Translate` in `machine/translate.cc`, where this method is used, and how `PageFaultException` can be thrown in this process.

Figure out how memory access is implemented and where the virtual to physical translation is performed in using the above method. You can trace the MIPS simulator (in `machine/mipssim.cc`) in executing instructions which trigger Page Fault.

It is during the translation process that the machine can determine if the virtual address it is trying to access belongs to a page which does not reside in physical memory. Figure out how, when and where the `PageFaultException` is thrown.

> Where is the page fault handled?

The page fault exception is handled in a manner similar to system calls in `exception.cc`. As a part of

raising the page fault exception, the MIPS machine saves the faulting virtual address in a register that will be used by `exception.cc` to call `VirtualMemoryManager::swapPageIn()`.

When is the process image stored on the disk and how is SWAP managed?

When a binary file is loaded, its process image including binary code and data sections is stored in the SWAP area.

For Project 3, SWAP is essentially a Linux file implemented through the stub interface of the Nachos file system (look into `filesys/filesys.h` and `filesys/openfile.h`). SWAP contains 512 sectors and each sector has 128 bytes with the same size of a memory page.

A swap bitmap (`swapSectorMap`) is allocated in `VirtualMemoryManager` to keep track the use of each sector of SWAP with a bit. Function `virtualMemoryManager::allocSwapSector()` is used by `addrspace.cc` to allocate a sector to store a page of the process image during binary loading.

How many memory pages are allocated to each process when it starts during initial loading (Nachos `-x` option), `Exec()`, or `Fork()`?

Zero. As this process executes an instruction or accesses a user-space data address, this generates page fault exception and triggers the page fault exception handling.

In Task 1, what are main steps of page fault handling in Function `SwapPageIn()`?

The number of C++ code lines you write for this task is within 50.

- Perform the second-chance algorithm as approximated LRU by checking the use bit of the next page.
- If the next page's use bit is `TRUE`, give a second chance by setting the use bit as `FALSE`.
- If the next's use bit is `FALSE`, you find a victim page.
- Check if the selected victim page is dirty, initiate I/O to write the contents of the victim page to SWAP.
- Adjust the pagetable for the process to which the victim page belongs to reflect the fact that it is no longer resident in memory.
- Initiate I/O to load the desired page for which the fault was generated from the backing store SWAP.
- Adjust the pagetable for the faulting process to reflect the fact that this selected physical page points to the new process space and the desired virtual page is now resident in memory.

What are main data structures and functions that can be leveraged in Function `SwapPageIn()`?

- The basic data structure manipulated by this function is page table entries. The fields of a page table entry defined in `machine/translate.h` of this starter code include the virtual page number of a process space (`virtualPage`), the physical page number (`physicalPage`), the use bit (`use`), the dirty bit (`dirty`), the valid bit (`valid`). The page table entry defined in the starter code also includes The offset of the corresponding sector in SWAP by bytes (`locationOnDisk`). We ask you to remove this storage location field and store such information in the process address space in `addrspace.h`.
- `FrameInfo` data structure for a physical page includes the process space this physical page belongs to and the virtual page number (`pageTableIndex`) in that space.
- Other key functions in `VirtualMemoryManger` include `allocSwapSector()` to get a free SWAP sector, `writeToSwap()` that performs disk I/O to write back a page to a sector in SWAP, `loadPagetoCurrVictim()` that loads the corresponding virtual page to the assigned physical page.
- `VirtualMemoryManager::getPageTableEntry(FrameInfo *physPageInfo)` returns the page table entry of process space that currently owns this physical page.

How to do Task 2?

The total number of line changes is expected to be small.

- Use `grep LocationOnDisk */*.cc` or compile after removing `LocationOnDisk` field in `machine/translate.h` to see where this filed has been used. Understand how it has been used in storing/loading a process page to/from the disk.
- Add a field in `AddrSpace` class that is initialized as a pointer to an integer array that gives the SWAP section location of each virtual page.
- Implement `putLocationOnDisk(int virtualPage)` and `getLocationOnDisk(int virtualPage)` methods.
- Replace the read/write access of `LocationOnDisk` field in all files with the above two methods.

How to print more debugging info?

Use `-d v` argument. For example, `../vm/nachos -d v -x testvmfork`. The output format is explained as follows. For the following outputs, `pid` is the id of the process on which behalf the operation is performed. `virtualPage` is the involved virtual page number (i.e. the page index into the process virtual address space) and `physicalPage` is the involved physical page number (i.e., the page index into the physical memory of the Nachos virtual machine).

1. Whenever a page is loaded into physical memory, print

   ```
   L [pid]: [virtualPage] -> [physicalPage]
   ```

2. Whenever a page is evicted from physical memory and written to the swap area, print

   ```
   S [pid]: [physicalPage]
   ```

3. Whenever a page is evicted from physical memory and not written to the swap area, print

   ```
   E [pid]: [physicalPage]
   ```

4. Whenever a process obtains a zero-filled demand page for the first time (i.e., when you allocate and zero the page out), print

   ```
   Z [pid]: [virtualPage]
   ```

Is synchronization needed for concurrent processes in executing system calls?

You should worry about synchronization issues. In general, two kernel activities such as system calls or exception handling may be handled concurrently and context switch is possible. For example, one process handles page fault, which triggers asynchronous I/O for dirty page disk writing or page loading and can sleep in the kernel while another process may initiate I/O on the same page at the same time. Synchronization such as locks can be needed for each memory page.

In Project 3, the concern is less as long as there is no explicit kernel yield or sleep involved to trigger context switching during the middle of a kernel service. Notice that Nachos disk I/O involved in Projects 2 and 3 uses the Linux file system in the lower level synchronously, which does not trigger kernel thread yield/sleep, and that is not true in a real OS.

Is TLB used in Nachos?

The current implementation disable s the TLB feature available in Nachos.